Efficient bulk-loading methods for temporal and multidimensional index structures

Dissertation zur Erlangung des Doktorgrades
der Naturwissenschaften (Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik
der Philipps-Universitat Marburg
vorgelegt von

Daniar Achakeev
aus Kharkov (Ukraine)

Marburg an der Lahn 2013

Erstgutachter:	Prof. Dr. Bernhard Seeger
Zweitgutachter:	Prof. Dr. Peter Widmayer

Zusammenfassung

Nahezu alle naturwissenschaftlichen Bereiche profitieren von neuesten Analyse- und Verarbeitungsmethoden für großen Datenmengen. Diese Verfahren setzten eine effiziente Verarbeitung von geo- und zeitbezogenen Daten voraus, da die Zeit und die Position wichtige Attribute vieler Daten sind. Die effiziente Anfrageverarbeitung wird insbesondere durch den Einsatz von Indexstrukturen ermöglicht. Im Fokus dieser Arbeit liegen zwei Indexstrukturen: Multiversion B-Baum (MVBT) und R-Baum. Die erste Struktur wird für die Verwaltung von zeitbehafteten Daten, die zweite für die Indexierung von mehrdimensionalen Rechteckdaten eingesetzt.

Ständig- und schnellwachsendes Datenvolumen stellt eine große Herausforderung an die Informatik dar. Der Aufbau und das Aktualisieren von Indexen mit herkömmlichen Methoden (Datensatz für Datensatz) ist nicht mehr effizient. Um zeitnahe und kosteneffiziente Datenverarbeitung zu ermöglichen, werden Verfahren zum schnellen Laden von Indexstrukturen dringend benötigt. Im ersten Teil der Arbeit widmen wir uns der Frage, ob es ein Verfahren für das Laden von MVBT existiert, das die gleiche I/O-Komplexität wie das externe Sortieren besitz. Bis jetzt blieb diese Frage unbeantwortet. In dieser Arbeit haben wir eine neue Kostruktionsmethode entwickelt und haben gezeigt, dass diese gleiche Zeitkomplexität wie das externe Sortieren besitzt. Dabei haben wir zwei algorithmische Techniken eingesetzt: Gewichts-Balancierung und Puffer-Bäume. Unsere Experimenten zeigen, dass das Resultat nicht nur theoretischer Bedeutung ist.

Im zweiten Teil der Arbeit beschäftigen wir uns mit der Frage, ob und wie statistische Informationen über Geo-Anfragen ausgenutzt werden können, um die Anfrageperformanz von R-Bäumen zu verbessern. Unsere neue Methode verwendet Informationen wie Seitenverhältnis und Seitenlängen eines repräsentativen Anfragerechtecks, um einen guten R-Baum bezüglich eines häufig eingesetzten Kostenmodells aufzubauen. Falls diese Informationen nicht verfügbar sind, optimieren wir R-Bäume bezüglich der Summe der Volumina von minimal umgebenden Rechtecken der Blattknoten. Da das Problem des Aufbaus von optimalen R-Bäumen bezüglich dieses Kostenmaßes NP-hart ist, führen wir zunächst das Problem auf ein eindimensionales Partitionierungsproblem zurück, indem wir die Daten bezüglich optimierte raumfüllende Kurven sortieren. Dann lösen wir dieses Problem durch Einsatz vom dynamischen Programmieren. Die I/O-Komplexität des Verfahrens ist gleich der von externem Sortieren, da die I/O-Laufzeit der Methode durch die Laufzeit des Sortierens dominiert wird.

Im letzten Teil der Arbeit haben wir die entwickelten Partitionierungsvefahren für den Aufbau von Geo-Histogrammen eingesetzt, da diese ähnlich zu R-Bäumen eine disjunkte Partitionierung des Raums erzeugen. Ergebnisse von intensiven Experimenten zeigen, dass sich unter Verwendung von neuen Partitionierungstechniken sowohl R-Bäume mit besserer Anfrageperformanz als auch Geo-Histogrammen mit besserer Schätzqualität im Vergleich zu Konkurrenzverfahren generieren lassen.

Abstract

The recent increase of spatial and temporal data requires efficient algorithms for index construction and for bulk updates. Many big data applications exhibit not only a high volume of static data but also inherit data growth. Moreover, some of them display high update rates. Incoming collected or produced data arrive in batches in order to reduce transportation and update costs. Therefore, updating an index using one record at a time is found to be inefficient for sufficiently large batch sizes. In this work, we investigate the problem of efficient bulk-loading of temporal and spatial index structures. We design novel loading strategies for multiversion B-tree (MVBT) and R-tree.

We introduce a novel loading algorithm for MVBT with the asymptotic optimal I/O complexity. We show that the previously developed technique based on the buffer tree solves the loading problem only for the special case (if an input file consists only of insert operations). In this work, we propose the first loading algorithm for MVBT that meets the lower-bound of external sorting. We also proposed an efficient algorithm for bulk updates. These results are achieved by a combination of two algorithmic techniques: buffer tree and weight balancing.

In this work, we designed a loading algorithm for the R-tree that optimizes it according to a widely used cost model. Extensive experimental results show that R-trees built using our novel approach exhibit substantially better query performance than sort-based counterparts. The novelty of our technique is that if a query profile is available the algorithm is able to build better R-trees. We proposed the following heuristic: first, we define the best sorting order using the average shape of the query rectangle. Second, we reduce the bulk-loading problem to a one-dimensional partitioning problem by sorting the input set. Third, we find an optimal solution for this problem according to the proposed cost models. The motivation of our heuristic is NP-hardness of the optimization problem. Based on the result obtained from query adaptive loading, we observed that construction of spatial histograms resembles the problem of optimal loading of R-trees. Therefore, we developed a spatial histogram construction method based on the partitioning framework developed for R-tree loading. Spatial histograms built using our novel method exhibit high accuracy for different data and query distributions. Our method has only one parameter, minimal page capacity. Moreover, our extensive experimental results show robustness of our method for different query and data distributions.

Acknowledgments

Foremost, I would like to thank my adviser Prof. Bernhard Seeger. I am very grateful for his guidance, invaluable advise, patience and encouragement.
I would like to thank members of the database group. I would like to express special thanks to Marc Seidemann for helpful comments, discussions and reviewing. I would like to thank Daniel Schäfer, Philip Schmieglt and Johannes Drönner for invaluable discussions and advise. I am very grateful to my RTM colleagues Michael Cammert, Christoff Heinz, Jürgen Krämer and Tobias Riemenschneider for their support and for the opportunity to work in a great team. I am thankful for the chance to work with Peter Widmayer on the problem of optimal bulk loading of R-trees. I would like to express my sincere gratitude to Ben Mills, Eugen Walter, Yannick Stein and Tobias Ebert for their helpful comments and discussions. I would like to thank Ben for his patience during the reviewing of this work.
I am extremely thankful to Anne Sophie Knöller for her invaluable support, for her understanding and patience.
Finally, I would like to thank my family. I am grateful to my parents Gulbara and Kulmuhamed, to my brother Emil and my sister Lia for all their love, for their encouragement and support.

Contents

1 Introduction

Large-scale data analysis allows new insights for research in fields of natural science as well as social science and humanities. Interdisciplinary research teams and practitioners use frameworks for big data analysis to understand and explain natural phenomena [117, 16], to predict, prevent and monitor natural and social catastrophes. As the majority of collected data has two additional dimensions space and time, temporal and spatial index structures as well as selectivity estimation methods are core components of large-scale analytic frameworks. They enable efficient query processing.

The recent increase of spatial and temporal data requires efficient algorithms for index construction and for bulk updates. Many big data applications exhibit not only a high volume of static data but also inherit data growth. Moreover, some of them display high update rates [60]. Incoming collected or produced data arrive in batches in order to reduce transportation and update costs. Therefore, updating an index using one record at a time is found to be inefficient for sufficiently large batch sizes. For one-dimensional index structures, there are efficient solutions to cope with high update rates and index bulk loading from scratch [49, 62, 41].

In this work, we investigate the problem of efficient bulk-loading of temporal and spatial index structures. We design novel loading strategies for multiversion B-tree (MVBT) [35] and R-tree [65]. Due to demand for efficient temporal data processing [75, 3, 2, 18, 78, 112], different temporal index structures have been proposed in the last three decades [105, 112]. Many of them are extensions of B+tree [82, 35, 116]. The multiversion B-tree (MVBT) [35] is the first partially persistent index structure with optimal worst-case guarantees for inserts, updates, deletes and temporal key-range queries. Recently, Haapsalo et al. [66] introduced recovery and transactional support for this structure. Thus, MVBT can be fully integrated in transactional systems. However, efficient algorithms for bulk loading and bulk updates are available for neither the MVBT nor for other partially persistent B-trees, e.g. the time-split B-tree [82, 85]. The design of efficient bulk algorithms for partially persistent B-trees is still challenging [82, 35].

The R-tree is an index structure for indexing multidimensional sets of rectangles [65, 38, 39] and is available in almost all modern database and geographic information systems.

There are many I/O efficient bulk-loading approaches for R-trees, e.g. [104, 73, 80, 58, 44, 27], yet all of them disregard knowledge of query profiles. We aimed to design a loading strategy that optimizes average query performance and considers query profile if it is available. If the query profile is not present then we minimize the volume of minimal bounding rectangles of the R-tree nodes according to cost models developed by [73, 115, 95]. In this work we show that statistical information about the average shape of a query rectangle can be used to generate better R-trees. Although the method proposed by [25] builds an R-tree with worst-case optimal query performance, this method is difficult to integrate into practical systems. In this work, we are interested in algorithms that exhibit low system integration cost and are conceptually simple. Therefore, we consider as a groundwork a sort-based R-tree loading approach.

We summarize our contribution as follows:

- We introduce a novel loading algorithm for MVBT with the asymptotic optimal I/O complexity. We show that the previously developed technique based on the buffer tree [24, 44] solves the loading problem only for the special case (if an input file consists only of insert operations). The general case (a mix of insert, delete and update operations) cannot be solved using this previous approach [44]. We also proposed an efficient algorithm for bulk updates. These results are achieved by a combination of two algorithmic techniques: buffer tree [27] and weight balancing [28].

- We designed a loading algorithm for the R-tree that optimizes it according to a widely used cost model [73, 115, 95]. Extensive experimental results show that R-trees built using our novel approach exhibit substantially better query performance than sort-based counterparts. The novelty of our technique is that if a query profile is available the algorithm is able to build better R-trees. We proposed the following heuristic: firstly, we define the best sorting order using the average shape of the query rectangle; secondly, we reduce the bulk-loading problem to a one-dimensional partitioning problem by sorting the input set; thirdly, we find an optimal solution for this problem according to the cost models proposed by [73, 115, 95]. The motivation of our heuristic is NP-hardness of the optimization problem [7].

  Based on the result obtained from query adaptive loading, we observed that construction of spatial histograms resembles the problem of optimal loading of R-trees. In both cases, a disjoint partitioning of input rectangles should be produced. Therefore, we developed a spatial histogram construction method based on the partitioning framework developed for R-tree loading. Spatial histograms built using

our novel method exhibit high accuracy for different data and query distributions. Our method has only one parameter, minimal page capacity. Moreover, our extensive experimental results show robustness of our method for different query and data distributions.

This work is organized as follows. In Chapter 2, we briefly review and introduce problem definitions, the computational model and multiversion B-tree and R-tree index structures. Chapter 3 presents related work where we describe the essential algorithmic techniques such as buffer trees and weight balancing in detail. A thorough understanding of these concepts is required in the following Chapter 4. In this chapter, we present a novel solution for loading partial persistent B-tree in asymptotic optimal I/O complexity. Chapter 5 introduces a solution for query adaptive loading of R-trees. Here, we present our novel partitioning framework for R-tree loading and show in Chapter 6 how it is used for construction of spatial histograms. At the end of Chapters 4, 5 and 6 we conclude with our results for the given techniques. Chapter 7 summarizes future work.

The major parts of this work were published in the following publications [4, 5, 6, 7, 8].

2 Preliminaries

In this section we review multiversion B-tree (MVBT) and R-tree index structures. We introduce important notations and the computational model. The focus of this work is the design of efficient loading techniques for MVBT and R-tree (see Chapters 4 and 5). We assume that the reader is familiar with concepts of tree-based index structures. We also refer the interested reader to very good surveys on temporal and multidimensional structures by Salzberg et al. [105] and Gaede et al. [57] as well as the book by Hanan Samet [106]. David B. Lomet presents in his work a detailed overview of index structures implementation approaches [83].

In this work, we consider two types of index structures: *partial persistent* and *ephemeral structures* [52]. In contrast to ephemeral structures (ordinary index structures), persistent structures manage both previous and current object versions. If the updates are allowed on any version we have a general concept of full persistence. For partial persistence, the previous versions are read-only and updates are allowed only on the most recently created version. The versions produced in the case of partial persistence are represented as a list and in the case of full persistence a version tree [52]. From a database perspective "version list" is equivalent to the notion of transaction time. The inserted items are not physically deleted. They are marked as deleted and are not accessible at the current state. Partial persistent index structures allow an efficient access either to past or current states of the database. [35, 26, 116]. Ephemeral structures such as B+tree or R-tree manage only the current state.

For brevity, we assume that the input is a sorted set of N triples $[ops, item, t_i]$. Triples are sorted according to $t_i \in \{1, \ldots N\}$ versions. After processing a triple, we transfer the index structure to a new state. The first element $ops \in \{insert, delete, update\}$ is a flag. We do not associate any particular algorithm with *ops* for the underlying structure. After the processing of a triple $[insert, item, t_i]$ an item is assumed to be present in all subsequent states of a data structure until it is deleted. Processing of $[delete, item, t_i]$ removes an item from the current state and *update* updates the item value in the current state. Thereby, we assume that the input set fulfills the following constraints:

- For each triple with a *delete* flag there must be a preceding triple with an *insert*

flag and same *item*.

- For each triple with an *update* flag there must be a preceding triple with an *insert* flag and same *item*.
- For each triple with an *insert* flag there are zero triples with the same item or the most recent triple has flag *delete* with the same *item*.

The type of the second element depends on the underlying index structure (e.g. rectangle or key-value pair). This allows us to introduce the following notions: *bulk loading*, *bulk update*, *bulk insert* and *bulk delete* independently from the underlying data structure. For example in the case of R-trees, items are axis parallel rectangles. Therefore, the bulk operation is defined as processing of N triples $[ops, item, t_i]$. The bulk processing algorithm uses flag *ops* and version number t_i to decide if the items have to be included in the current state or not. In the following sections, we will see that this corresponds to a partial persistent file. We introduce problem definitions of bulk operations from this perspective. Note that this is an abstract definition of the problem and must not correspond to a real physical representation. Yet it allows us to define problems in a very generic way, as we assume that we can convert each item from the real input set in a constant time.

- *Bulk loading* is the problem of constructing an index structure from scratch using N triples $[ops, item, t_i]$.
- *Bulk update* is the problem of processing N triples $[ops, item, t_i]$ on a non-empty index structure. Additionally, we define the problem of processing N $[insert, item, t_i]$ or $[delete, item, t_i]$ triples on non-empty index structures as *Bulk insert* or *Bulk delete*, respectively.

The first problem that we tackle in this work is an efficient bulk loading and bulk update of the MVBT structure. In this case the assumption of the input set is fully compliant with our generic definition. In the second problem addressed in this work, we study R-tree bulk loading. The input set of loading problem, in this case, consists of triples with *insert* flag only. As an R-tree is an ephemeral structure and all flags are *insert*, after bulk loading all data is present in a current state. Therefore, we can use arbitrary ordering for efficient loading of R-trees. Hereafter we use the term ***loading*** to refer to the creation of an index for a given input.

2.1 I/O Model

The RAM (random-access machine) model does not sufficiently reflect the performance of I/O dominant algorithms [17]. The main downside of this model regarding external memory algorithms is the unified memory assumption. The complexity of the algorithm is expressed in the number of instructions. Each instruction is processed by CPU in the same amount of time. The model assumes a random access unbounded memory layout. The time penalty of accessing any memory location is identical [17]. This model is more or less realistic for small problem sizes that can be processed in main memory. In I/O dominated external memory algorithms, I/O requests take considerably more time than other instructions [17]. Moreover, the modern hardware architecture implements memory as a hierarchy. On top we have fast but small sized CPU caches as well as main memory and on the bottom we have magnetic disks, SSD and tapes with significantly larger capacity. Their access time is also orders of magnitude larger in comparison to cache or main memory access. The actual processing time of the algorithms is often dominated by disk accesses [17]. In this work we will use the I/O model by Aggarwal and Vitter [13] as a default measure of the algorithm complexity. Further, we will use the notion CPU costs for costs obtained using the RAM model for problems that can be processed in memory.

In the I/O model, the memory hierarchy is simplified by two-level memory architecture. The main memory is a volatile bounded random access memory with an access time orders of magnitude smaller than external memory access. External memory is non-volatile random access memory. For brevity, hereafter we call this memory the disk. It is partitioned in fixed size blocks (or pages) that are the smallest transfer units between external and main memory. The problem size considered in the I/O model is assumed to be substantially larger than the size of the main memory. The model also assumes a single CPU. However, it allows reasonable performance comparison to external memory algorithms. In this work we assume that only one block at a time is transported from disk to main memory (this is complaint with the assumption in [13, 119, 118]). At the beginning of the algorithm we always assume that the data is located on a disk.

We express the size of the main memory in the number of records M. A disk block contains B records. By fetching a single block from a disk, we always transport B records from disk to main memory. Table 2.1 summarizes notations used in this work. We assume that the input problem occupies $n = N/B$ blocks on a disk. The space complexity is also expressed in the number of blocks needed on a disk to solve the problem. In the following we assume that $2 \cdot B \leq M$ and $M << N$.

Symbol	Description
N	problem size (in number of records)
M	memory capacity (in number of records)
B	block capacity
$m = M/B$	memory capacity (in number of blocks)
$n = N/B$	problem size (in number of blocks)

Table 2.1: Important notations

In their seminal work [13], Aggarwal and Vitter showed non-trivial lower bounds of external memory algorithms. One of the major results is that the least number of block transfers for external sorting is equal to $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. Other important bounds are the worst-case I/O bound for searching in unsorted file is $\Theta(\frac{N}{B})$ and the worst-case I/O bound for searching in a sorted file is $\Theta(\log_B N)$, e.g. using B$^+$tree.

The I/O model does not ideally reflect the processing time of the algorithms in real world applications. For example, for a magnetic disk there is a substantial time difference for accessing the set of blocks randomly or sequentially. Therefore, algorithm engineers should also consider hardware characteristics for performance improvement. In addition, emerging non-volatile memory technology such as solid state disk or flash devices pose new challenges for algorithm engineers [15], since there is no gap between random and sequential access time, yet read access is in general faster than write. Nevertheless, the I/O model is still a good performance indicator for the I/O intensive external memory algorithms.

Another interesting computation model that encapsulates memory hierarchy is the cache-oblivious model by Frigo et al. [55]. On different memory levels data is transported in different units. For example, from disk to main memory it is about 4KB and between CPU cache and main memory it is multiple machine words. Parameters such as available main memory M, block size B are tuned in order to achieve the best performance of external memory algorithms in practical application [17]. Additional knowledge about cache sizes is also used to obtain the best results. External memory algorithms need an explicit knowledge of particular hardware characteristics. In contrast, the goal of the cache-oblivious model is to allow algorithm engineers to design algorithms that exhibit optimal numbers of block transfers on any memory hierarchy without tuning the parameter M and B for each memory level.

As in the I/O model [13], it assumes a two-level memory hierarchy. Additionally, it assumes that the data moved between the memory levels using optimal replacement strategy in memory units of the particular level, and caches, are fully associative [55].

The asymptotic optimal cache-oblivious algorithm exhibits an optimal number of cache-misses and an asymptotic optimal number of data movements on any memory hierarchy [40, 55]. Regardless of the optimal replacement strategy and fully associative cache assumption, algorithms developed for this model achieve very good performance in real world applications such as write-optimized index structures for one-dimensional range searches [41].

2.2 Partially Persistent B-tree (MVBT)

In this section we tackle the problem of managing records in a partially persistent file consisting of multiple versions. Partial persistence is a well-known concept in computational geometry [107, 61, 26]. In databases partial persistence is used to manage old object versions [84, 122] and to enable efficient history querying. Recently, partial persistence has also played a key role in developing robust transactional file systems [102] and key-values stores [1].

Here the term version describes a record given by the following tuple $< k, t_s, t_e, inf >$ where k is a key. $[t_s, t_e)$ represents a version interval in which the key is valid, and inf is the payload. A versioned record is alive in the most recent version if its t_e field carries the special character "*". Otherwise the versioned record is dead. Versioned records can be depicted as intervals in a two-dimensional space, consisting of a time dimension (x-axis) and a key dimension (y-axis). The i-th version of the partially persistent file consists of all versioned records $< k, t_s, t_e, inf >$ with $i \in [t_s, t_e)$. Update operations are allowed only on the most recent version, but queries are supported on any version. Whenever an update operation (insert, delete) is posed, a new time stamp now is created and a new record with version interval $[now, *)$ is inserted (in the case of insert) or a live record is deleted (in the case of delete). Note that a deletion corresponds to closing the interval of a live entry by assigning now to the t_e field. An update on a versioned record is simply a concatenation of insert and delete (without incrementing the version number before delete).

Due to the excellent worst-case performance, a partial persistent B-tree, e.g. MVBT [35], is used as underlying structure for supporting queries on any version. The leaves of the MVBT consist of versioned records. In addition, the version concept is also carried over to the index entries, i.e., an index entry also comprises a time interval $[ts_i, te_i)$.

MVBT (multiversion B-tree) is an asymptotically optimal partial persistent B$^+$tree. It has $O(N)$ space complexity and supports (key range) queries at version i with the same asymptotic complexity as an ordinary B$^+$tree that only stores the i-th version.

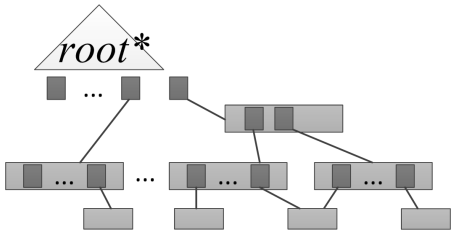


Figure 2.1: MVBT structure

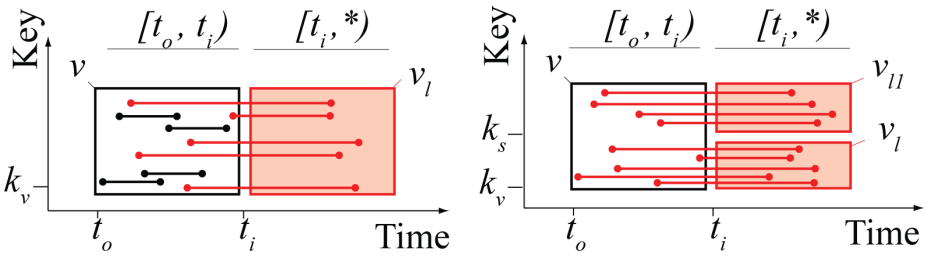
I/O time for the i-th update is $O(\log_B N_i)$. Construction of MVBT is performed by update and requires $O(N \log_B N)$ I/Os in the worst case. MVBT is actually a direct acyclic graph (DAG), providing a condensed physical representation of N B$^+$trees (one for each version) [35]. As proposed by Discroll et al. [52], MVBT stores pointers to historical roots in a separate (B$^+$tree) termed *root**. The DAG and root* of an MVBT are illustrated in Figure 2.1.

The asymptotic bounds on query and update time are achieved by preserving the so-called *weak-version condition*: a linear fraction of the capacity $d = \frac{B}{4}$ in a live node is reserved for live data. The remaining portion $\frac{3B}{4}$ can be used for historical (dead) entries. For the sake of simplicity, we use these specific settings throughout this thesis without loss of generality. We refer to [35] for a detailed discussion on parameter settings.

Reorganization of a live node is triggered if there are not enough live entries in the node (i.e., the weak version condition is violated) or the physical capacity B is exceeded. In order to use only linear space, the so-called *strong-version* condition has to be satisfied: the number of live entries is to be between $\frac{3B}{8}$ and $\frac{7B}{8}$ for nodes that have just been involved in a reorganization. Therefore, such a node accepts at least $\Theta(B)$ updates (insertions, deletions) before its next reorganization will be triggered.

We discuss the specific reorganization operations of MVBT using the four two-dimensional partitionings of the time-key space shown in Figures 2.2 and 2.3. Each (leaf) node of an MVBT corresponds to a rectilinear rectangle. Intervals represent the versioned records; black and red ones refer to dead and live entries, respectively. We assume that an update at version t_i triggers a reorganization. Reorganization of a node always starts with a time split where live entries at version t_i are copied from node v to a new live node v_l (see Figure 2.2(a)). If the strong version condition is violated for v_l, additional reorganization steps are triggered. If v_l has more than $\frac{7B}{8}$ live entries, a *key-split* is performed first; see Figure 2.2(b). Similar to a split in a B$^+$tree, entries are evenly distributed among two nodes using a split value from the key dimension.

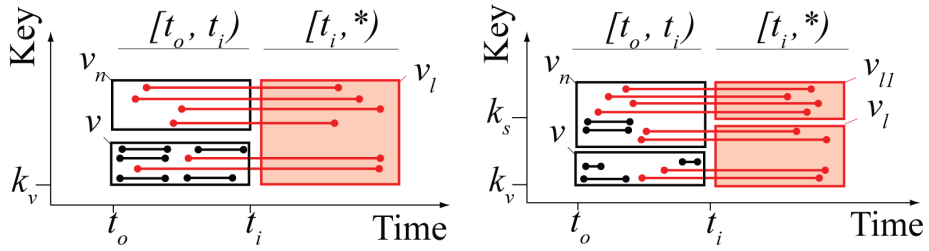
If v_l contains fewer than $\frac{3B}{8}$ live entries, a *merge* with a live key sibling node v_n is



(a) Time split (b) Key split

Figure 2.2

triggered. We can find live siblings by accessing the parent node. After a time split on v_n, live entries from v_n are inserted into v_l, as seen in Figure 2.3. If the number of live entries in v_l is greater than $\frac{7B}{8}$, an additional key-split has to be performed on v_l. This is illustrated in Figure 2.3(b). Thus, the two new live nodes satisfy the strong-version condition. Hereafter we use the term *node reorganization* to refer to time split, merge or key-split. Note that at most two new nodes can be created during one reorganization.



(a) Merge (b) Merge-Key-Split

Figure 2.3

Algorithm 1 describes the insert procedure of MVBT given a record $e =< k, inf >$ at time t_s. The path to a live leaf node is computed in lines 1–4. In each level *chooseSubTree* searches for the matching live index entry using key k. Afterwards, the versioned record is inserted in the leaf node. If either the weak version condition or the capacity constraint is violated, a reorganization of the leaf node will be triggered (lines 5–7). After the reorganization process is finished, we update time interval(s) of the deleted node and insert the new live index entry or entries in the parent node (lines 8–18). Then we recursively check if a reorganization of the parent node is needed.

If the live root is reorganized, a new live root is created (lines 10–15) and a corresponding index entry is inserted in *root**. In the case of an additional key-split, the height of the live MVBT tree increases (lines 12–13). Otherwise, the height remains the same and

the live root is replaced by its temporal successor (line 14–15).

We can arrange entries in an MVBT node either according to the key or to the time dimension. Sorting according to the time dimension seems to be the natural choice, since we can simply append new records after reorganizations. The search cost during the insert procedure is linear in B. Additionally, in the case of a split we also need to find a median in a current live set with at least linear CPU costs. Alternatively, we can manage two lists, one for dead entries and one for live (this would not increase the amount of space of the node needed for entries on a disk, as only a constant amount of memory is needed to represent both lists such as number of elements). With this second arrangement only the dead entries list is appended and sorted according to the time dimension with the live list being sorted by the key dimension. If a key is deleted we remove it from the live list and append it to the dead list. This would enable us to search the node with only logarithmic CPU cost. Additionally, logical delete of an entry can be implemented using logarithmic CPU cost, for example using a balanced search tree for live entry list. This organization also improves CPU costs of key-splits and a sibling search in the case of merge operation. Although the node layout does not influence theoretical I/O cost, we can improve CPU performance in practical applications.

The root structure is organized as a B^+tree. We use end time stamp of the time interval to index dead root entries. This allows us to efficiently insert entries into root*, since the entries are indexed by the order of deletion time. We append a deleted root entry to the rightmost leaf node of the tree. Thus, we build root* bottom-up with amortized I/O cost of $O(1)$ per deleted root entry. root* yields a time partitioning of the input set. These are then used for an efficient query processing on old versions.

In order to support efficient key-range queries over a given time interval a leaf is linked with its predecessor(s); see line 7 in Algorithm 1. Every leaf manages up to two backward pointers to its temporal predecessor(s). This allows us to start the processing at the right border of the search rectangle and to traverse backwards through the leaf level. First, the root is determined that is responsible for the most recent version of the time-range. Thereafter, we find all leaves within the key-range that belong to the most recent version of the time-range. Starting from these leaves, we use the backward links to locate all the other required leaves. The details for query processing are given in [50].

Haapasalo et al. [66] introduced transaction and recovery algorithms for MVBT. They called their structure transactional multiversion B-tree (TMVBT). In order to support transactions on MVBT node reorganization algorithms were modified. The authors show that TMVBT allows one write and multiple read transactions at a time while maintaining all worst-case performance guarantees of MVBT. They developed their algorithms

for multiversion concurrency-control protocol (MVCC) [45]. MVCC assigns versions incrementally to a transaction, such that transactions are allowed to see the data of transactions that were committed before. All the records updated by the transaction are assigned the version number of this transaction. To this end, the first modification of an MVBT is to support multiple records with the same version.

As only one write transaction at a time is allowed, TMVBT nodes are partitioned into groups. The *active nodes* are alive, contain data of a current write transaction and their start time is equal to the version of the write transaction. The *non-active nodes* contain data of committed and uncommitted transactions. TMVBT enforces the rule that active nodes have only a single parent. The authors introduced modified reorganization operations for active and non-active pages. For example, for the set of active pages standard B+tree algorithms are used, while for non-active nodes modified MVBT node reorganizations are applied. All records of the current write transaction are copied to a new live node and deleted from the dead node. Since there are two types of nodes, merge operations between two type of nodes are also introduced. We refer the reader to [66] for details of TMVBT. Therefore, MVBT can be fully integrated in modern transactional database systems.

2.3 R-tree

The R-tree index structure owes its popularity to its conceptual simplicity, good average query performance and its broad application field. Since the seminal work of Antonin Guttman [65], R-tree has attracted researchers both from theoretical and applied fields. In the last three decades, scientists and engineers have done large amount of work to improve the R-tree query performance. There are several milestones in the research history of R-tree, for example the popular R-tree variant R*-tree [38], the first R-tree with asymptotically worst-case bound on window queries [25], cost models for query performance prediction [73, 115, 95], and sort-based as well as top-down bulk-loading algorithms [104, 73, 80, 58, 44, 27].

An R-tree is a balanced tree, proposed by Guttman et al. [65], for indexing d-dimensional set of rectangles. Figure 2.4 displays an R-tree built for a set of two-dimensional axis parallel rectangles. In the I/O model nodes of R-tree are mapped to pages. Therefore, except for the root node, R-tree nodes have a capacity between B and minimum occupation $b \leq \lceil B/2 \rceil$. All leaf nodes are on the same level. Leaf nodes contain input records. This can be object reference (disk address, RID or TID) or the object itself. Additionally, we provide a function that maps input record to an axis

Algorithm 1: Insert

```
   Input: Entry e, Time Stamp ts
 1 cR ← root;
 2 while cR does not point to leaf do
 3 |   node ← GetNode (cR) and push node in Path;
 4 |   cR ← ChooseSubTree (node, e, ts)//search live index entry;
 5 InsertLeaf ( leaf ← GetNode (cR) , e, ts);
 6 idx[] ← SplitNode ( leaf, ts)// perform reorganization if needed, if merge or
   merge-key-split then find neighbor;
 7 link created successor nodes with leaf;
 8 while idx[] is not null do
 9 |   get parent node, logically delete cR, parent ← pop Path;
10 |   if parent is null then
11 |   |   store cR as historical root ;
12 |   |   if was key-split then
13 |   |   |   create new root node and insert new live successors idx[] in it;
14 |   |   else
15 |   |   |   replace root with new created live successor;
16 |   else
17 |   |   insert new live successors idx[] in parent;
18 |   |   idx[] ← SplitNode ( ParentNode, ts);
```

parallel rectangle. Further, record sets belonging to leaf nodes are disjointed. An index entry stores a node address and a *minimal bounding rectangle (MBR)*. The MBR of an internal node is a union of child node MBRs. The leaf node MBR is computed over the set of leaf records using data-dependent map function. Without loss of generality, we assume that this function is the identity such that the input records are axis parallel rectangles or MBRs.

The basic query types supported by R-tree are point and multidimensional range query (aka window query). The point query computes all record MBRs that contain a d-dimensional query point q_p. A window query computes, for a given axis parallel query rectangle q, all record MBRs that overlap query q. Since q_p is a special rectangle, we treat point query as a special case of window query. To answer a window query we start at the root of the tree. We compute, for example in depth first manner, all child rectangles that overlap q. We process this step recursively until all overlapping nodes have been visited. In the worst case all nodes can be visited. Therefore, a query should visit only nodes that contribute to the query result. The method of node MBR generation influences the

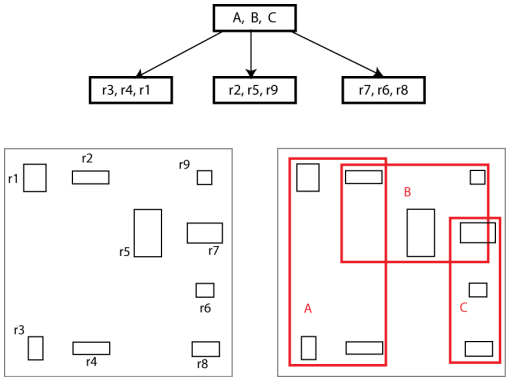


Figure 2.4: An R-tree with page capacity $B = 3$ and minimal page capacity $b = 2$ build for a set of rectangles $r_1, \ldots, r_9$

query performance. Guttman [65] noticed that node MBR area should be minimized to achieve good average query performance, since this would minimize the probability of overlap with a query rectangle. Later, this was confirmed by [73, 115, 95].

The authors in [12, 74, 68, 25] show that the lower I/O bound for a given window query is $\Omega((\frac{N}{B})^{1-1/d} + r/B)$ where r is the number of results. However, in practical application R-tree exhibits better average I/O cost. The average good query performance is owed to sophisticated heuristics applied during updates on the R-tree [65, 38, 39, 104, 73, 80, 58].

The single update operation delete or insert is applied in top-down fashion. The worst-case insert and delete cost is equal to $O(\log_B N)$. Generic insert procedures start at the root of the tree. Then the best child is chosen for descent according to some heuristic. These steps are repeated recursively until a leaf node is reached. If applicable, node MBRs on the path are adjusted, for example if the node MBR only partially contains the record MBR. Guttman in [65] proposed selection of a child entry such that its MBR would have the smallest area increase. In the case of tie breaks a child node with the smallest MBR area is chosen. Beckmann et al. [38] also consider the sum of the overlap areas.

Similar to B$^+$tree, node overflow is handled by a split. However, the way the node records are distributed between the two new nodes has a larger impact on query performance. The goal of a split algorithm is to partition $B + 1$ entries into two partitions such that each partition has at least b entries. Not all possible splits are good. We define the MBR of a partition as union of entry MBRs of this partition. Guttman proposed generating such partitions that minimize MBR area [65]. He introduced two algorithms, one with a linear CPU complexity $O(B)$ and one with a quadratic $O(B^2)$. The idea of both algorithms is to find two extreme MBRs, so-called seeds. They will build two

partitions. The quadratic algorithm finds two MBRs with the largest value of the area of their union MBR subtracting the two MBR areas. Then it assigns the remaining rectangles to the partitions in a manner similar to inserting algorithm; for example, entry is assigned to a partition with a minimum enlargement of partition MBR area. The linear algorithm finds two extreme MBRs for all dimensions with the highest lower left point and lowest upper right point along the particular dimension. Then the distance between extreme corners is normalized by the side length of MBR computed over $B + 1$ entries for a particular dimension. The pair with a highest value are chosen as seeds [65].

Beckmann et al. [38] introduced the most popular split variant used in modern database systems. It has $O(B \log B)$ CPU complexity and their variant of R-tree called R*-tree exhibits very good query performance. The idea is to sort MBRs for each dimension and quantify $(B - 2b + 2)$ possible splits for each dimension. The MBRs are sorted twice for each dimension using the lower left and the upper right corners. The authors proposed three metrics to quantify the split quality: 1. the sum of partition MBR area, 2. the perimeter of partition MBR and 3. the overlap of partition MBR. Their method is a heuristic, yet R-trees built with their method display good query performance in comparison to counterparts. Becker et al. [34] proposed algorithms for computing optimal split partitioning according to one of these metrics with a CPU complexity equal to polynomial with a degree proportional to the number of dimensions. They also derive a lower bound $\Omega(B \log B)$ for a computation of this partitioning.

The authors in [38] also considered minimizing perimeters of the MBR. This gives a preference to square-shaped node MBRs resulting in compact representation of tree nodes, since for a given area a square shaped rectangle minimizes the perimeter [38, 57]. Beckmann et al. [38] investigated different strategies for defining the best split. The following strategy exhibits the best result. Firstly, the best dimension is defined. The best dimension is one with the smallest sum of perimeter metric (all possible splits are considered for this dimension). Among the splits on this dimension the one with the smallest overlap is chosen; in the case of tie breaks area metric is considered. The split is then performed using a hyper plane perpendicular to the sorting dimension. The authors also proposed other techniques for query performance improvements such as forced reinsert. Instead of performing a split in the case of node overflow a fraction of node entries are reinserted in the R-tree. We refer the interested reader to [38, 57] for details. In practical applications this option is often skipped due to high CPU and I/O costs as well as negative impact on concurrent transaction processing.

Delete operation finds records using its MBR and if it is present deletes from a leaf node. If applicable, node MBRs on the way from leaf to root are adjusted. However, in

the case of node underflow, in contrast to B^+-tree, delete operations in R-trees originally proposed by Guttman [65] do not execute merge or share reorganization. If a leaf node has fewer than b entries we remove its index entry from its parent. Then the remaining entries are added to a "re-insert" set. We recursively go up the path and check for an underflow. At the end of the procedure, we insert all entries from the re-insert set in a tree using insert procedure. We place entries in their original levels. Since this operation causes CPU and I/O overhead, we opt for local merge and if applicable subsequent split operation instead (or we apply the global rebuilding). The merge neighbor is defined using MBR of the underflow node as if we insert this MBR in the parent node. We refer to Bercken et al. [42] for implementation details. The works by [76, 77, 67] introduce algorithms for transaction and recovery support for R-trees.

3 Related Work

In this section we review bulk-loading techniques for multiversion B-trees and R-trees. Loading algorithms can be roughly classified into three groups: tuple-by-tuple, bottom-up loading and top-down loading. First, we present the core generic loading algorithms. Afterwards, we discuss related work for specialized loading approaches for MVBT and R-trees. We refer the interested reader to Jeffry Vitter's book [118] as well as the survey by Arge et al. [30]. They present many algorithmic techniques for external memory used in this work.

Although more I/O efficient loading techniques exist, the simplest loading method is to execute *insert* procedure for each input object [49, 35, 65]. If the time complexity of the insert procedure is $O(\log_B N)$, then N input objects are inserted using $O(N \log_B N)$ I/Os. Hereafter we call this type of loading *tuple-by-tuple*. Both R-tree and MVBT can be loaded using this approach. In practical applications, time efficiency of this method can be improved by utilizing a page (node) buffer. However, the improvement rate depends on page replacement policy and on an ordering of insert operations.

A more I/O efficient technique, known from B-trees [49, 63], is to build an index in bottom-up fashion, starting from the leaf nodes. For now, we assume that an input data set is sorted according to an appropriate criterion. A core variant of this technique creates a leaf node and its associated index entry for each B (or a fraction of B) elements from a sorted set. In the next step, these index entries are used as an input set for the next index level generation. Index entries are processed according to the order of their generation in the previous step. These steps are recursively repeated until fewer than B elements remain. Finally, a root node is created. The loading I/O time is equal to $O(N/B)$. Each iteration reduces the input set by a factor B. However, if the data is not sorted, we need at least $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ [13] I/Os for sorting. Thus, the overall I/O costs are dominated by the I/O cost of external sorting. We consider *sort-based bottom-up loading* as a two-step approach: first, we sort the input data and afterwards level-by-level recursive construction is applied. Sort-based bottom-up loading of B-trees reduces the build time by at least factor B compared with tuple-by-tuple solution. The advantage of this technique is its conceptual simplicity. Moreover, since an external

sort algorithm is available in most database systems, we regard implementation and integration costs as very low. This approach is also applicable to R-trees. It has been found that a sorting according to space-filling curves leads to R-trees with good query performance [104, 73].

Unfortunately, this technique is not directly applicable for bulk-loading MVBT. In partial persistence we need to preserve strict temporal order of input records. Constructing MVBT can be seen as a two-dimensional sweep line algorithm, where at each event on the x-axis (time axis) one of the operations *insert, delete* and *update* is executed [61]. Although input entries are already sorted according to their time stamps, we need to manage a dynamic ordered set of live keys in order to build correct MVBT. However, we can still use the tuple-by-tuple approach that corresponds to the described sweep line analogy. However, we are at least a factor B away from our desired complexity. To tackle this problem, we adapt top-down loading technique based on buffer trees [22, 23, 24, 44, 27].

3.1 Buffer Trees

Arge developed an external memory data structure called *buffer tree* for off-line (batched) problems that efficiently utilizes available memory of size M [22, 23, 24]. The general idea is to process and push down elements in batches. For this purpose, buffers of size equal to the portion of available memory are attached to the internal nodes. This technique transports $\Theta(M)$ elements with I/O costs of $\Theta(M/B)$ between two levels. This yields $\frac{1}{B}$ amortized I/O cost per transported entry.

The buffer tree algorithm empties the buffer only after they are filled completely, in order to amortize I/O cost for a set of operations. In contrast to a B$^+$tree, a record is pushed towards the leaf node "lazily" after several buffer emptying processes. This defers execution of a single operation such as insertion of an entry. To solve off-line problems efficiently, delete and update of an entry are also processed "lazily". Lars Arge modeled this by attaching the time stamp of the operation t and operation type $ops \in \{insert, delete, update\}$ to an entry managed by the buffer tree. Hereafter we assume records have the following format $< ops, k, inf, t >$: k is a key of the record, t time stamp of operation, *ops* operation type and inf information payload (see Section 2). For brevity, we consider *insert* and *delete* operations.

One of Arge's main results is that input data can be sorted using an optimal number of I/Os only using the insert procedure of the buffer tree. In this way, the technique is also applicable for problems where the complete input is not present at the start of loading,

since the input is processed in sufficiently large batches iteratively. He also proposed and devised I/O-efficient solutions based on buffer trees for off-line (batched) problems in the fields of computational geometry and graph problems. Later, the buffer tree ideas were used for designing I/O-efficient loading approaches for R-trees by Bercken et al. [44] and Arge et al. [27].

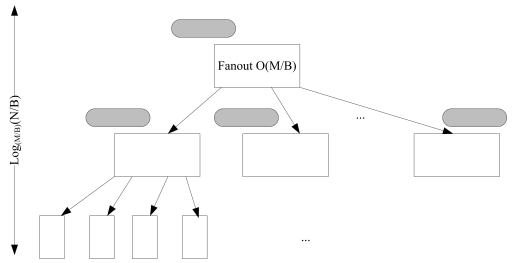


Figure 3.1: Buffer Tree Architecture [23]

The basic variant of the buffer tree is a height balanced (a, b) search tree [23, 69] (as shown in Figure 3.1). Each internal node has a fanout between $a = M/4B$ and $b = M/B$ (number of routing elements). However, leaf nodes have a capacity of B elements. Hence, the height of a tree is equal to $O(\log_{M/B} N/B)$. Each internal node has a buffer of capacity M/B pages attached. Additionally to leaf and internal nodes, we define the parent nodes of leaf nodes as "leaf buffer nodes". As a buffer tree is an (a, b)-tree [69], violation of a node capacity invariant is repaired by a series of rebalancing operations such as *fuse*, *share* and *split*. If a node has fewer than a children, depending on the child number of neighbor node either a fuse or share operation is executed. The fuse merges two neighbor nodes. The share operation redistributes entries between two neighbors. If a node has more than b children the split is then executed creating one more node. As the buffer tree manages buffers, rebalancing operations are slightly modified. We will now review the insert algorithm of the buffer tree presented in [23].

The insert procedure starts at the root of the tree. A buffer is defined as full if it contains more than $M/2$ records. After collecting B records in memory in a single block, if a root buffer is not filled completely, we append a block to a buffer. Otherwise, we trigger the buffer emptying process. Except for nodes that reference leaves, we first load $M/2$ records into memory and sort them according to the key. As both delete and insert operation with the same key can be present in the same buffer, we delete corresponding insert delete pairs (according to their time stamps). Then we iterate through the record sequence and append them to child node buffers according to the keys. Since the number of child nodes is $\Theta(M/B)$, we load one non-full block in the memory of each child buffer and $\Theta(M/B)$ index entries associated with child nodes. If all child buffer blocks are full

we allocate a new one. For all child nodes with more than $M/2B$ full buffer blocks we start the buffer emptying process recursively. In contrast to a root buffer, the buffer of an internal node can contain more than $M/2$ records, for example, all records of a parent buffer are pushed to single child buffer. The cost of buffer emptying on internal nodes is bounded by $O(M/B)$.

We do not empty "leaf node buffers" until all of the internal nodes with full buffers above are processed. The buffers of "leaf node buffers" are completely emptied. The buffer emptying process sorts the buffer records and deletes corresponding insert delete pairs. Let k be the number of leaf nodes of node v. The process loads and merges leaf node records with sorted buffer records in a single list of blocks.

Then for each corresponding block we create a leaf node. At this point, routing elements are modified in the parent node. If the number of resulting leaves is greater than k then split rebalancing is executed. If applicable we continue splits on the way from the leaves to the root. The buffer of the internal node can be redistributed between the new and old node. The I/O cost of split rebalancing is bounded by $O(M/B)$ I/Os, as the node buffers on the path from new leaves to root have no more than $M/2$ records.

If the number of resulting leaves is less than k, then a list of "dummy" blocks (that correspond to deleted leaf pages) is created such that the number of leaves and dummy blocks is k. Dummy blocks of node v are processed one by one, executing the following steps. The buffer emptying process deletes one dummy block of v and checks balancing condition on v. In contrast to (a, b)-tree rebalancing, the buffers of the neighbor node and the violated node buffer are emptied before share or fuse operations. If applicable, rebalancing is applied on the way from v to root. As we push buffer records of sibling nodes, we cannot prevent buffer overflow of other leaf buffer nodes. In this way, other buffer leaf nodes could overflow due to buffer emptying. In this case, we execute the emptying process on this node first. Therefore, after deleting one dummy block we need to wait until all rebalance operations are complete, in order to synchronize buffer emptying processes.

Recall, since buffer leaf nodes are processed only after buffer emptying processes on internal nodes are finished, internal node buffers are not full (have fewer than $M/2B$ blocks). Each rebalancing operation has costs bounded by $O(M/B)$, since we need to process M/B buffer blocks of two nodes, modify routing entries and post changes to a parent node.

Arge showed that a buffer tree sorts N records using asymptotic optimal numbers of I/Os. To sort the data using a buffer tree the data is loaded into an empty buffer tree and then all buffers are emptied in a breadth first manner. The cost of loading of N

records is bounded by $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. The cost of buffer emptying is $O(\frac{M}{B})$. On each node on level l we pay $O(\frac{M}{B})$ I/Os after each $M/2$ operation. I/O costs for all levels are then $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. The cost of any rebalancing operation is also bounded by $O(\frac{M}{B})$. As the buffer tree is an (a, b)-tree, the number of rebalancing operations is bounded by $O(N/B \div M/B = N/M)$ [23, 69]. Thus, the I/O cost of all rebalancing operations is $O(N/B)$. The emptying of all buffers is bounded by $O(N/B)$, as there are $O(N/M)$ buffers and single buffer emptying I/O cost is bounded by $O(M/B)$.

Inserting N records in a buffer tree and subsequent emptying of all buffers produces a sequence of sorted blocks. The block size is $O(B)$. This fact is used by Bercken et al. [44]. The authors developed a generic loading algorithm using the buffer tree of Arge. The index structure for example the R-tree is built recursively level by level. The loading approach first builds the leaf node level using a buffer tree technique in the same manner as if it were used for sorting. The associated index entries of the leaf nodes are then used for index level generation. These steps are repeated until fewer than B records remain. At this point, we create a root node of the index. The proposed algorithm is applicable for loading only; bulk updates are not supported.

In each step, a modified buffer tree is used for constructing the current level's entries for the index structure. The loading algorithm loads entries into an empty buffer tree and subsequently empties all buffers. The buffer tree uses a routing algorithm of the underlying structure for pushing elements one level down. The leaf node level is built using $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. At each round of the loading, the input set of a buffer tree is reduced by a factor B. In this way, we obtain a geometric series for the overall I/O cost that is also bounded by $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$.

Their loading approach can be applied for a wide range of index structures. However, loading of MVBT is possible for the limited case of insertions only. We discuss problems of buffer trees for MVBT loading later in Chapter 4.

Arge et al. [27] proposed an improved version of the buffer tree loading algorithm for R-trees. This supports loading as well as bulk insertion, deletes and updates. In contrast to buffer trees [23, 44], it does not use nodes with a high fan-out ($\Theta(m)$). Their loading approach solves the loading problem more elegantly by attaching the buffers into internal nodes of the R-tree as and when required. This allows us to execute efficient bulk operations without changing the node layout of the tree. Due to these buffers, the R-tree insert procedure is slightly modified. Their approach follows the same idea of processing elements in batches. Conceptually the proposed structure is a buffer tree with a fan-out $\Theta(B)$ and buffer size $M/2$. However, the buffers are attached to the nodes on levels $i \cdot \lfloor \log_B \frac{M}{4B} \rfloor$.

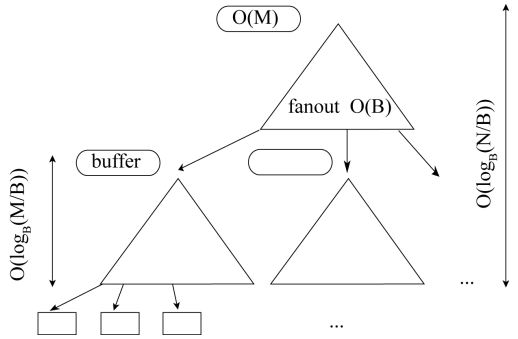


Figure 3.2: Buffer tree by Arge et al.

Figure 3.2 depicts the architecture of the proposed method. The basic idea is to to load a sub-tree in memory to route records from a buffer to the next buffer level. Thus, records are pushed $\log_B \frac{M}{4B}$ levels down without any I/O. As in a buffer tree, buffers are emptied only after they are completely filled. Except for the lowest buffer nodes, buffer emptying pushes records in batches of size $M/4$. This guarantees that there is no buffer with more than $M/2$ records present. The lowest buffer nodes (nodes on level $\lfloor \log_B \frac{M}{4B} \rfloor$) are always completely emptied. The sub-tree of height $\lfloor \log_B \frac{M}{4B} \rfloor$ has $O(M/2B)$ nodes. Thereby, a buffer size $M/2$ and a sub-tree fit in main memory of size M. The I/O costs of buffer emptying is equal to $O(M/B)$ I/Os, since we load $M/2B$ buffer pages and $M/2B$ nodes for routing.

The loading process is similar to buffer tree loading. However, the R-tree routing algorithm is used. As in the buffer tree, rebalancing is executed in a bottom-up fashion after emptying lowest buffer nodes. For bulk loading an R-tree, only the split rebalancing operation is considered. Similar to the buffer tree, the split operation on buffer node redistributes buffer content between old and newly created buffer nodes.

The I/O cost of split operation is again bounded by $O(M/B)$ I/Os. The overall R-tree bulk loading is executed in two steps. N spatial elements are inserted into the R-tree with buffers. Then all buffers are emptied. Arge et al. showed that this technique builds an R-tree using the same number of I/Os as external sorting. Similar to buffer tree cost, it follows from buffer emptying and rebalancing costs.

The advantage of the new buffer technique is support for bulk updates, inserts and deletes. For brevity, we review bulk insert and delete procedures only. Bulk insert is a minor modification of a bulk-loading procedure. Instead of starting with an empty tree, bulk update attaches empty buffers to an existing R-tree. Then N records are inserted in an existing tree similar to the first step of bulk loading. After the last entry is inserted in a root buffer, all buffers are emptied in a breadth first manner. Let N' be a number

of records present in R-tree before bulk insert, then the worst-case I/O cost is equal to $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N'+N}{B} + \frac{N'}{B})$.

With R-tree bulk loading and bulk insert, emptying the lowest buffer nodes is handled recursively. There is no need to wait before all above buffer nodes are emptied, since only split rebalancing is considered. After buffer emptying of the lowest buffer node each buffer contains at most $M/4$ records on the way to it; redistributing of buffer contents does not trigger buffer emptying processes. Bulk delete on buffer R-tree introduces a merge rebalancing operation. After delete operation reaches the leaf nodes, changes are posted to the parent nodes. If the buffer node violates the capacity condition it is merged with its neighbor. The neighbor is computed using the routing algorithm for R-trees. Additionally, for buffer nodes buffer content is also merged. Merging of buffer nodes could also trigger buffer emptying, as two neighbor nodes can contain $M/4$ records. In this case the buffer emptying processes could interfere with each other. Arge proposes modification of buffer emptying on the lowest buffer nodes. Similar to a buffer tree [23], lowest buffer nodes are emptied only after all buffer nodes above are emptied. After emptying all upper buffer nodes, we process buffer emptying of the lowest buffer nodes one by one. If rebalancing causes buffer overflow on upper levels then all affected buffer nodes are emptied before the next lowest buffer is processed. Arge also presents in his work other bulk operations; we refer the interested reader to [27].

The buffer tree technique [22, 44, 27] solves the index-loading problem for a wide range of index structures with the same asymptotic costs as for external sorting. In the case of loading, in general, they introduce implementation and execution overhead due to buffer management, bookkeeping of buffer emptying processes and bottom-up rebalancing operations. However, buffer tree [27] is applicable for a wide range of bulk operations such as insert, update and delete. Moreover, authors propose hybrid methods mixing sort-based and buffer tree techniques [27]. Recently, the buffer tree technique has been adapted for the design of efficient index structures optimized for flash and solid state drives [15].

In our work, we consider both techniques; we propose novel sort-based loading approach for R-trees. The algorithm takes a query profile into account to build better R-trees according to a widely used cost model. For MVBT we designed a novel loading algorithm. It uses the buffer tree of Arge et al. [27] as a groundwork. However, we designed a novel buffer tree technique that not only solves the loading problem efficiently, but also gives other advantages related to implementation complexity and concurrency.

3.2 Weight Balancing

Our approach for loading MVBT uses a combination of buffer tree and weight balancing technique. Weight balancing is an algorithmic technique and is used for example to amortize the costs of reorganization operations on data structures. Mark Overmars presents generic algorithmic techniques for the design of data structures in his excellent work [94]. Firstly, we present the application of this technique in the case of partial rebuilding of search trees. Let us consider a binary search tree structure that enforces some node balancing condition using local rebalancing operations (rotations). In some cases, it is beneficial to defer (or not to execute) local rebalancing operations and rebuild the sub-tree of a highest node that ran out of balance [94]. Sub-tree rebuilding is done by constructing a perfectly balanced tree (using the sub-tree nodes) in a bottom-up fashion. This can be done in a linear number of operations. Weight-balanced binary search trees proposed in [91] allow us to amortize the cost of partial rebuilding. The average insertion cost of such trees using this approach is logarithmic.

To the best of our knowledge weight balancing was first proposed by Nivergelt and Reingold [91]. They introduced a class of balanced binary search tree BB[α]-tree. Let $w(v)$ be a number of nodes in a tree with root v (inclusively v). Further, we define v_l and v_r as a left and right child of v. They define the balance $\rho(v) = \frac{w(v_l)}{w(v)}$ of tree node v by the fraction of the number of nodes in a left sub-tree of v to the number of nodes in a sub-tree with root v [91, 94]. The node balance is bounded by parameter α. Each node of a BB[α]-tree fulfills balance constraint $\alpha \leq \rho(v) \leq 1 - \alpha$. The tree has a logarithmic height for $\alpha > 0$. If the node balance is violated due to insert or delete operation, local rebalancing operations such as rotation or double-rotation are performed. Blum and Mehlhorn showed that for $\frac{2}{11} \leq \alpha \leq 1 - \frac{\sqrt{2}}{2}$ [46, 94] at most two rebalancing operations are needed after insert or delete.

Although the worst-case time to insert an element is bounded by $O(\log N)$, Overmars showed that for $0 < \alpha < \frac{1}{2}$ BB[α]-trees the average cost of insert is equal to $O(\log N')$ operations using the partial rebuilding technique. N' denotes a maximal number of entries in a tree at some point of time [94]. If a sub-tree with root v is perfectly balanced then $\Omega(w(v))$ inserts and deletes can be performed on a sub-tree of v before it violates the weight condition. As rebuilding costs are also bounded by $\Theta(w(v))$, we obtain the upper bound. This can be shown using the accounting method. Every time an entry passes through the node we put one cost unit to a bank account to pay a future rebuilding. Per operation we put at most $O(\log N')$ cost units [94]. Therefore, BB[α]-tree can be built without implementing standard rebalancing operations very efficiently.

Inspired by the ideas of partial rebuilding on BB[α]-trees, Lars Arge and Jeffrey Vitter [28, 29] designed an elegant structure: weight-balanced B-Tree (WB-tree) for the I/O model. Their goal was to design an I/O efficient dynamic structure for interval managing. The proposed external interval tree efficiently answers *stabbing queries* [54]. For a given set of intervals a stabbing query returns all intervals that contain a query point [29]. Using WB-tree Arge et al. achieve amortized worst-case bound for update operation on the dynamic version of an external interval tree.

WB-tree is a weight-balanced variant of the B-tree. WB-tree maintains weight information for each node. In contrast to B-tree, the insert procedure additionally updates node weights. Node splits are triggered based on the node's weight information. For now we consider insertion only.

The weight $w(v_l)$ of a leaf node is defined as the number of elements stored in it. Function $p(v)$ returns the parent node of node v. The weight of internal node v is defined recursively as $w(v) = \sum_{(v=p(c))} w(c)$ the sum of its child weights. Thereby, internal node weight $w(v)$ is equal to the number of elements in descendant leaves of v. The ratio of the node weights on the same level is constant. Node weight $w(v)$ increases exponentially with each level.

WB-tree guarantees that the split operation (rebalancing) on node v occurs after $w(v)$ insert operations. In this way, if the rebalancing operation costs are bounded by the node weight $O(w(v))$ then the amortized cost is equal to $O(1)$. We use this crucial property for developing our solution, since this allows us to estimate the least number of operations needed before the next rebalancing operation occurs. For example, in combination with a buffer tree we estimate the number of operations that can be pushed down without violation of MVBT conditions.

In the following we review WB-tree in detail. Node weight $w(v)$ is defined using the so-called branching parameter $a > 4$. WB-tree leaves are on the same level and have weights between k and $2k - 1$. Internal node v on level l has a weight less than $2a^l \cdot k$. Except for the root, the internal node on level l has a weight larger than $1/2a^l k$. The root has more than one child.

Crucial properties of WB-tree are (the interested reader is referred to [28] for details): 1. Except for the root, all nodes have between $a/4$ and $4a$ children. The root has between 2 and $4a$ children. 2. The height is equal to $(\log_a(N/k)$. 3. After a split of node v on level l into two nodes v_1 and v_2 , at least $a^l k/2$ inserts have to be performed below v_1 (or v_2) before it splits again. After a new root r in a tree containing N elements is created, at least $3N$ insertions have to be performed before r splits again.

A WB-tree can be implemented on top of a B-tree, for example by attaching a weight

counter to index entries. Although this technique is conceptually simple, we need to slightly modify the insert and split procedures. Firstly, the insert procedure updates node weights. We increase the weight counter of a node by one during the descent from a root to a leaf. This introduces a constant I/O overhead in comparison to I/O complexity of B-tree insert algorithm. Secondly, the split procedure is also modified. Let us consider node v on level l. The split decision is made using child weights instead of the number of entries. We iterate through the sequence of child entries (sorted according to the key domain) and sum up their weights. We stop iteration if the current sum is greater than $\frac{1}{2}w(v)$ (half of the node weight). Then, we split node v at the child position where we stopped. The WB-tree properties guarantee that for branching parameter $a > 4$ an almost balanced split can always be computed.

Delete operation on a WB-tree is implemented using the global rebuilding strategy [29, 94]. We illustrate the key idea of global rebuilding using the following example. We refer interested readers to [94, 29] for details. Using this global rebuilding we achieve worst-case bounds for delete operation without handling local rebalancing. Instead of physical deletion we mark elements as deleted. Say we allow at most $N/2$ elements to be deleted before we reorganize the tree. After a certain fraction of elements are marked as deleted, say $1/4N$, we start to construct a second tree lazily containing $3N/4$ elements from the first one. We still use the first tree for queries as well as insert and delete operations. However, for the following $1/6 \cdot (3N/4)$ insert/delete operations we insert 6 elements in a new tree using the insert procedure. Additionally, we store these new operations. They need to be performed on a new structure after it is completely built, since we cannot use the new structure until these operations are executed. We use the next $1/6 \cdot (1/6 \cdot (3N/4))$ operation on the old structure to insert $1/6 \cdot (3N/4)$ temporarily stored operations with continuing to store incoming operations. This process continues recursively until all temporarily stored operations are executed on the new structure. At this point the new structure can be used and the old one discarded. Using these parameters it can be shown that only one tree is under construction at a time, since the last temporarily stored operation is executed within $1/4 \cdot (3/4N)$ operations [94, 29].

Arge et al. uses WB-tree to amortize split costs of proposed interval tree resulting $O(\log_B N)$ insert costs. One variant of their interval tree is a WB-tree with a branching parameter a equal to $\frac{\sqrt{B}}{4}$ and leaf capacity $2k = B$. Similar to the main memory interval tree [54], interval end points are used to index intervals. Nodes have associated secondary structures that contain up to $w(v)$ intervals. For a detailed discussion of the external interval tree we refer to [29]. A split cost and subsequent update of secondary structures of new and old nodes as well as posting changes to a parent is bounded by $O(w(v)/\sqrt{B})$

I/Os. Due to the fact that at least $O(w(v))$ has been executed on the node since the last split, the cost of a split is amortized.

Gioara and Kaplan also use a WB-tree variant for developing an I/O efficient solution of the vertical ray shooting problem [59]. Recently, weight balancing has been used for devising efficient cache-oblivious index structures, e.g. cache-oblivious B-tree [40]. As in Arge's interval tree, weight balancing is used for amortizing the reorganization costs.

3.3 Multiversion B-tree Loading Algorithms

To the best of our knowledge, the first solution for bulk loading a partial persistent B-tree in I/O model was proposed by Goodrich et al. [61]. They built a persistent B-tree with branching degree $\sqrt{\frac{M}{B}}$ in I/O complexity of external sorting to solve a geometric off-line problem. To solve the problem they used the *distribution sweeping* paradigm. The main idea is a recursive partitioning of a set of N operations in $s = \sqrt{\frac{M}{B}}$ stripes of roughly N/s size. This setting allows us to hold $\sqrt{M \cdot B}$ elements of each partition in memory. Data is partitioned according to the y-axis (key dimension) in s partitions at each recursive step in a top-down fashion, since we represent data as horizontal intervals in two-dimensional time-key space. No key partitions overlap. Partitioning steps are applied until a partition fits in the main memory. Afterwards, a persistent B-tree with a fan-out s is built on this partition. Each recursive call returns a list of historical roots of s-way persistent B-trees. Note that each list is sorted according to the time stamp. Afterwards, each s-th element from root lists is taken as a *bridge* element and merged in a single result list Y. The merging procedure is done by recursively constructing lists of bridge elements. Additionally, pointers to all bridge predecessors in recursively constructed lists are stored with bridge elements. The list Y together with pointers defines roots for a persistent structure [61].

To find an element alive at time point i an alive root at time i is located. The structure is then traversed in a top-down fashion always searching in nodes whose time stamp is the largest value smaller than or equal to i. In contrast to MVBT, the time for locating an element at time i in this structure depends on the value of $s = \sqrt{\frac{M}{B}}$. Our problem differs in that we address the loading of an online persistent B-tree with branching degree B, a problem that is so far unsolved [61].

Bulk loading of the MVBT was already addressed by Bercken et al. [44]. The authors used a generic buffer tree framework [24, 44, 27]. Although this approach is applicable to loading R-trees, loading of MVBT is only possible for insertions. For mixed workloads consisting of insertions, deletions and updates, this approach cannot be used. We will

show major problems of this approach in Section 4.2. Our new loading algorithm is applicable to arbitrary workloads while all asymptotic performance guarantees of the MVBT are fully maintained.

An interesting loading algorithm for R-trees is presented in [27] (see Section 3.1), using the buffer tree framework. This algorithm loads the leaf level and index levels simultaneously. The advantage is that this approach is not limited to bulk loading only, but also suitable for bulk update. Unfortunately, the loading approach cannot be used for partially persistent B-trees. However, our new bulk-loading algorithm also loads all levels simultaneously. Therefore, our loading algorithm can also be used for bulk updates as well, with very few changes.

Recently, Zhang et al. [122] presented a memory optimized tuple-by-tuple on-line loading algorithm for the HV-tree, an advanced version of the Time-Split B-tree (TSBT) [85]. The primary goal was to provide fast access to recent data in memory and to move old data efficiently to secondary storage. In contrast to our problem, the HV-tree assumes that all live nodes can be kept in memory. This assumption is not always valid, as the size of databases can still be larger than the available main memory. As a consequence, no worst-case performance guarantees are given. In addition, the loading algorithm still relies on executing one update at a time, while our approach achieves substantially higher improvements of the bulk update time from processing updates in batches.

Due to the continuously growing size of a versioned database, distributing this data among multiple nodes is becoming more and more important. In [79], a new method is presented for determining splitters for a set of versioned records (represented as intervals in a two-dimensional space). This method could be easily combined with our loading algorithms to obtain a distributed loading technique. In this work, however, we focus on the centralized case and leave a detailed discussion on distributed techniques for future work.

3.4 R-tree Loading Algorithms

The most generic method for loading R-trees is to apply standard insertion algorithms to each of the input rectangles. The loading time is then $O(N \log_B N)$, while the query performance solely depends on the underlying insertion algorithm. Insertion algorithms are designed in such a way that a goal function should be optimized for a split. In [95, 73], a cost model was introduced revealing that the perimeter and the area are the two crucial performance indicators. However, [34] shows that an optimal split of a node does not lead to globally optimal R-trees. This cost model provides the basis for our

investigations.

Roussopoulos and Leifker [104] introduced the problem of loading an R-tree from scratch and presented a sort-based loading technique with complexity $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, where M denotes the available main memory. After sorting the rectangles according to a one-dimensional criterion, an R-tree can be built bottom-up as it is known from B^+trees. Because the sorting order has a considerable impact on the search efficiency, Kamel and Faloutsos [73] proposed a double transformation: first a rectangle is mapped to a multidimensional point and then a space-filling curve like the Hilbert curve is used to generate a one-dimensional value. In order to improve query performance, heuristics like the one proposed in [51] can be used for local data reorganization.

STR [80] is also a sort-based loading algorithm that is conceptually different from the simple sort-based algorithms mentioned above. d different sort and partitioning phases are used, one for each dimension. The partitions after the last sort correspond to the leaf pages of the target R-tree.

The advantages of sort-based loading strategies are their simplicity of implementation yet a good query performance. Therefore, they are the only methods currently used in DBMS and GIS. However none of these methods can guarantee the quality of the generated R-tree regarding a cost model.

The Top-down Greedy Splitting (TGS) bulk-loading method [58] constructs the tree in a top-down manner by applying cost-optimized binary splits in a greedy manner. The cost function with the best experimental results [58] minimizes the area of the bounding boxes. The partitioning is performed by iterative binary steps where in each step multiple sorting orders are examined to detect the split with minimum area. In [25], it has been confirmed in experiments that the average search performance of R-trees generated by TGS are almost always better than the ones generated by other loading methods. Only for artifical data sets with highly varying aspect ratio has the priority R-tree been superior to TGS. A main disadvantage of TGS is its high loading cost (due to the binary partitioning) that can be substantially higher than the cost for external sorting. Due to its binary steps, it is difficult to parallelize TGS in a scalable manner. Other top-down partitioning techniques like QuickLoad [43] avoid expensive binary partitioning steps, but the design of an efficient parallel version is still an open problem.

Loading techniques based on buffer trees [44, 27] can be considered as a hybrid of top-down and bottom-up strategies. The basic idea is to delay insertions by temporarily storing input rectangles in buffers attached to the nodes. If buffers are filled up, the batch of insertions is reactivated and the rectangles continue their traversal down to the

leaves. In order to achieve better search quality it is suggested using a sort-based loading strategy for buffer emptying above the leaves. While the loading efficiency is the same as for external sorting, the underlying split algorithm (except for the leaf level) determines the query performance.

The priority R-tree (PR-tree) [25] is the first loading method whose target R-trees provide worst-case guarantees for window queries, while the loading can be performed with the same complexity as external sorting. It also has been shown [25] that the practical performance of the PR-tree is also good for two-dimensional data. However, in most cases it is not as good as for the R-trees of TGS, which is the only cost-model-sensitive loading technique so far. In fact, the PR-tree is not primarily designed for improving the average-case performance according to a cost model and a query profile. Moreover, its high implementation complexity might prevent it from being considered in a real system.

The theoretical foundations of the loading problem have been addressed in [96], where the NP-hardness of the bucket optimization problem has been proven, but only for a specific artificial cost function that substantially differs from the ones that are commonly used for R-trees [95, 73, 115]. This is contrary to our work, where NP-hardness is shown for the cost function [95, 73] minimizing the area of bounding boxes.

None of the previous methods have been designed for *query-adaptive loading* of R-trees. Query-adaptive loading refers to the problem of generating R-trees whose average performance is minimized regarding a given static profile. This is in contrast to adaptive indexing techniques like splay-trees [110] and database cracking [70], which apply structure adaptations during runtime of the queries. Different adaptive R-trees have been proposed in the literature [36, 114, 37], but all of them require a mix of queries and insertions to obtain the full benefits of adaptivity.

One of our approaches to query-adaptive loading relies on space-filling curves (SFCs) to obtain a one-dimensional ordering of the rectangles and on an optimal assignment of rectangles to pages. Most of the other approaches to using SFCs for sorting multidimensional data like [73] shuffle the bits in a symmetric manner, which is most suitable when every dimension provides the same selectivity. Orenstein and Merett presented a more flexible framework for shuffling bits that allows the definition of different sorting orders [93]. Based on their framework, we present shuffling strategies that adapt to the underlying query profile. A theoretical foundation for generation of query-adaptive space-filling curves was developed in [31], but without considering the specific problem of bulk loading. In addition to sorting, we also address the problem of data partitioning over a set of pages. The common packing strategy [73] to fill up pages to the maximum

leads to suboptimal query performance. In contrast, our new partitioning strategy relies on the dynamic programming framework used for generating optimal histograms [72].

The dynamic programming scheme proposed in [72] is also used in [120] for computing a set of k minimal bounding rectangles (MBR) from a two-dimensional point set. The goal was to reduce communication costs for mobile devices by approximating the spatial query result by a set of MBRs with a minimal information loss f_i. The authors showed that computing such representations is NP-hard even for d=2. One of their heuristics first sorts the query output using the Hilbert order and then applies the partitioning method of [72]. In contrast to bulk loading, space constraints are disregarded. In this work, we show that these constraints allow for the design of more efficient algorithms.

4 MVBT$^+$ Loading Approach for Multiversion B-trees

In this chapter we study bulk operations on partial persistent structures in external memory and address offline and online problems. The first problem is bulk loading: for a given operation set the goal is to build an index efficiently. The second problem is efficient support of bulk operation on an existing index. Our objective is to design an algorithm for a multiversion B-tree (MVBT) [35] construction with the same I/O complexity as external sorting. To the best of our knowledge, this problem has not been solved so far.

Unfortunately, design of sort based bottom-up solution similar to the B-tree loading is difficult, as the strict temporal order of operations needs a dynamic management of an ordered key set for each time stamp. Therefore, we consider other techniques such as buffer tree loading [24, 44, 27]. This generic technique assigns buffers to the nodes and builds trees in top-down fashion through the sequence of buffer emptying processes. Data is moved down towards leaves lazily in batches.

Although, the early version of MVBT loading approach [44] uses buffer trees. It has several downsides. In fact, it is applicable for insertions only. For general case of mixed workloads (insert, update and delete operations), it does not solve the bulk-loading problem correctly. Due to the temporal disorder of structure reorganizations, the invariants of the MVBT cannot be maintained anymore. Moreover, the problem of bulk-insertion was not addressed. In this work, we present an efficient algorithm for a bulk-loading MVBT that meets the lower bound of external sorting. The novelty of the approach is a combination of the buffer tree [27] and weight balancing [28] technique. The buffer tree technique allow to process data in batches using available memory efficiently. We use weight balancing to maintain MVBT constraints and to synchronize buffer emptying processes and structure reorganizations according to the temporal order of operations. Moreover, the new solution also supports bulk updates and can be used for online problems.

This chapter is organized as follows: In Section 4.1, we introduce preliminaries and

important notations. In Section 4.6, we describe a class of partial persistent B-trees that can be constructed in the asymptotic optimal time. Before we present theoretical results in Section 4.6, we outline the basic idea of our new bulk loading algorithm in Section 4.2. In addition, we show that it is impossible to use the original buffer tree for loading the MVBT. In Section 4.3, we present the details of our asymptotically optimal bulk algorithms for a concrete partial persistent B-tree. We outline the theoretical results on runtime for this tree in Section 4.3.3. In Section 4.4 we present our algorithm for bulk update. We discuss our experimental result in Section 4.7. Section 4.8 concludes the chapter.

4.1 Preliminaries

In the following, we consider the bulk loading and bulk update problem for the MVBT. Let us consider a sequence of input records $e_i =< k, inf, ops >$, $1 \leq i \leq N$, where $ops \in \{insert, delete, update\}$. For $i = 1, \ldots, N$, the operation of e_i is performed on the most recent version of the MVBT using tuple $< k, inf >$ as input. For bulk loading we assume the initial tree to be empty, while the tree already consists of N' live records for the bulk update problem. These problems differ from the equivalent ones on ephemeral indexes, e.g. B-tree and R-trees, in the sense that the partial persistent semantics requires a strict ordering of the update operations. In fact, this renders a direct application of traditional loading techniques impossible.

Symbol	Description
N'	number of live records
N	problem size (in number of operations)
N_i	number of entries live at version i
M	memory capacity
B	block capacity
d	minimal live entries per block

Table 4.1: Important notations

For the MVBT we use the following notation: N is the number of updates, N_i denotes the number of records live at version i. Parameter $d \in \Theta(B)$ denotes the minimum number of live records in a page. We count tree levels $l = 0, 1, \ldots,$ bottom-up starting from the leaf level. Our notations are summarized in Table 4.1.

4.2 Basic Ideas of Bulk Loading

In this section, we outline our approach to bulk loading a partial persistent B-tree, which is closely related to MVBT. Our goal is to provide a loading solution that requires $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os. In fact, this is the lower bound for loading because external sorting and loading of ordinary B-trees cannot be faster.

In order to design an efficient loading algorithm, we use two techniques in combination with the MVBT. The one is the *buffer tree* technique [27] (see Section 3.1) and the other is *weight balancing* [28] (see Section 3.2). We call this MVBT extension MVBT$^+$ because it maintains the worst-case performance properties of MVBT and additionally supports efficient loading in asymptotically optimal number of I/Os. Each of these two techniques contributes to the efficiency of MVBT$^+$:

1. The buffer tree yields the same I/O time as external sorting. The key idea of this technique is to transport data in batches between levels (see Section 3.1 and Figure 3.2).

2. Weight balancing controls the synchrony of buffer emptying processes. It guarantees that the MVBT$^+$ still maintains the MVBT invariants without giving up its worst-case performance.

The buffer tree attaches buffers to the nodes on each $i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ with $i = 1, 2, \ldots \Theta(\frac{\log_B N/B}{\log_{B/4} M/16B})$ level, see Figure 3.2. The buffer size is limited to $\frac{M}{2B}$ pages. This allows us to keep all live nodes of a sub-tree of height $\log_{B/4} \frac{M}{16B}$ in memory using at most $\frac{M}{2B}$ I/O. In Section 4.3.2 we explain the choice of these parameters in detail. We use the following terminology hereafter: a leaf (node) is the node on level 0, an index node is on level $l > 0$, a buffer node is the node on levels $i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ with $i = 1, \ldots$. A buffer is associated to a buffer node.

4.2.1 The Problems of Buffer Trees

In the following, we show that is not sufficient to use the buffer tree only for loading a MVBT. In fact, synchronization of buffer emptying is required; As reorganizations within the MVBT are temporally ordered according to the time stamps of the updates. If the buffers are emptied only after they are filled completely, some subtrees will evolve uncontrolled in time. Moreover, the node reorganizations of the child nodes can force the parent node also to evolve in time. This becomes a serious problem when other child nodes still contain historical data in their buffers. When these buffers are emptied later

in time, the parent node as well as siblings could already be dead. As a consequence, an insertion of their index entries in the parent node is not allowed, and a required merge with a sibling is impossible. In both cases, the MVBT invariants are violated.

The first problem termed *parent-child problem* is illustrated in Figure 4.1. Initially, there are one parent node p and two child nodes u and v. A time-split of u also leads to a split of parent p. The new parent node p' is created with time interval $[t_8, *)$, while its child v is already vaild at time t_2. Later the buffer emptying of node v creates two nodes v' and v''. The time interval of v' is $[t_4, t_6)$ which does not fit to the time interval of parent node p'. Therefore, it would be required to insert an index entry (referring to v') into the dead node p. However, an insertion into a dead node is not allowed for MVBT.

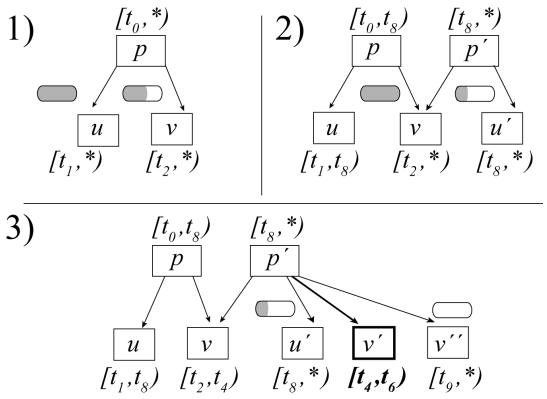


Figure 4.1: Parent-Child problem

The second problem termed *sibling problem* is illustrated in Figure 4.2. Initially, there are one parent p and two child nodes u and v. Further, node u is a key sibling of v. Node u evolves much faster than its sibling v. This causes a time-split at t_{10} and a new node u' is created with a time interval $[t_{10}, *)$. Later in time, the buffer of node v is emptied. Due to the historical data in the buffer, a time split is performed at time t_3. Therefore, the interval of v is closed and a new node v' with a time interval $[t_3, *)$ is created. Because v' contains less than $\frac{3B}{8}$ records, a merge is triggered with a node that is alive at t_3. This would be the key sibling node u, but u is already dead. However, a merge with a dead node is not allowed for the MVBT.

4.2.2 A Case for Weight Balancing

In order to avoid the parent-child problem and the sibling problem, we applied the weight balancing technique [28]. The idea is to prevent these problems even without knowing

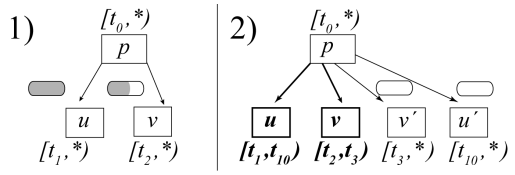


Figure 4.2: Sibling problem

the precise closing time of an entry time interval. Instead, we introduce a so-called *safe interval* where the closing time is estimated by the lower-bound of the number of operations required until the next reorganization will happen. Only if there is no overlap among two safe intervals (belonging either to siblings or to a parent and child), an additional reorganization step will be triggered. The larger the safe intervals, fewer of these forced reorganizations are necessary. Similar to B-tree, MVBT requires only $\Theta(B)$ operations to trigger the next reorganization after the last was performed. Unfortunately, this causes very short safe intervals and many forced reorganization steps. In contrast, weight balancing allows much more operations until the next reorganization step has to be triggered. More precisely, the number of operations is asymptotically equal to the number of records in the associated subtree. This leads to very long safe intervals and a very low probability that there is no overlap among safe intervals. Moreover, when a buffer of a subtree has to be emptied and the time interval of the associated entry moves to the future, the safe interval of the sibling is forced to overlap. If necessary, the buffer of the sibling has to be emptied first.

Similar to [28], weight information has to be maintained for each node. However, we use two different weight counters for a node v. The live weight $w(v)$ tracks the live records in the associated subtree (including its root buffer), whereas the operation weight $t(v)$ tracks the number of update and insert operations. The live weight is used to preserve the MVBT invariants, e.g. weak version condition. The operation weight $t(v)$ allows to estimate the closing time of the safe interval.

Both weights capture the temporal progress of a node. We constrain the ratio of node weights on the same level by a constant. With each level the weights $w(v)$ and $t(v)$ increase exponentially. Further, at least $O(w(v))$ operations are needed to reorganize node v on level l again.

Weight balancing requires a weight information attributed to each node and maintained during the loading process. This results in the following modifications of the original buffer tree emptying process:

- Weight information is updated during buffer emptying process.

- Buffer emptying is triggered either if the buffer is full or the weight conditions of the node are violated.
- Buffer emptying is forced if the node safe intervals do not overlap. However, as we apply weight balancing, I/O costs will not asymptotically increase.
- Node reorganization is performed in a top-down manner only. By this, the new entries can be stored in the parent node without causing any overflow again. This facilitates the implementation of the buffer tree (in comparison to its original counterpart).

Due to this top-down node reorganizations and emptying of buffers, we avoid *parent-child* problem and the *sibling problem*. As a consequence, the loading time of our approach is asymptotically optimal. The details are given in the next section.

4.3 Bulk-Loading Details

In this section we explain our MVBT$^+$ bulk loading methods. We first discuss the loading process using three procedures *Bulkload*, *ClearBuffer* and *PushDownEntry* (see Algorithms 2, 3 and 4). We also discuss how to apply the weight-balancing technique to loading. The main theorem proof is outlined at the end of the section.

Algorithm 2: Bulkload

```
   Input: InputData
1  initialize S, rootBuffer, root points to first leaf node;
2  foreach Entry e in InputData do
3      if rootBuffer size ≥ M/4 then
4          for i = 1 to M/4 do
5              PushDownEntry (dequeue entry, root, null, S, ts(entry));
6          foreach idx in S do ClearBuffer (idx) ;
7      append e to rootBuffer;
8  ClearAllBuffers ();
```

4.3.1 Buffer Tree Loading

Loading starts at the root of a tree (see Algorithm 2). Data is pushed towards the leaf nodes in batches if either the root buffer is full or a node weight condition is violated. MVBT$^+$ points to a buffer of the current live root (see line 1). The batch size is $\frac{M}{4}$ (see

line 3). Buffer nodes pointers with buffer sizes greater than $\frac{M}{4}$ are stored in stack S. Buffers of the buffer nodes stored in S are emptied right after the root buffer. Finally, function *ClearAllBuffers* (line 10) is called to empty the buffers of all buffer nodes in breadth-first manner (level-by-level).

Algorithm 3: PushDownEntry

```
   Input: Record e, Index Entry idx, Parent Node v_p, Stack S, Time Stamp ts
1  cR ← idx ;
2  if cR violates weight condition then
3      splitType ← ComputeSplitType () ;
4      if cR is not root and has buffer then
5          remove cR from S and ClearBuffer (cR) ;
6          if splitType is merge or merge-key-split then
7              nR ← find neighbor of cR;
8              remove nR from S and ClearBuffer (nR);
9      idx[] ← SplitNode ( v ←GetNode (cR) , v_p, ts, splitType) ;
10     if splitType is key-split then
11         reassign cR depending on key split ;
12     if v_p is null then
13         new root node handling ... // see Algorithm 1 lines 10–15 ;
14     else
15         ExpandParent (idx[]);
16 UpdateWeight (cR);
17 if cR has buffer and buffer size ≥ M/4 and not yet in S then
18     push cR in S ;
19 if cR is not root and has buffer then
20     Enqueue (e, buffer of cR) ;
21     return;
22 if cR points to leaf node then
23     load leaf node and insert entry e into leaf node;
24     return;
25 v ←GetNode (cR) and cR ← ChooseSubTree (v, e) ;
26 PushDownEntry (e, cR, v, S) //recursive call ;
```

For the $\frac{M}{4}$ entries of the root buffer *PushDownEntry* is called, see Algorithm 3. This function has the following parameters: record to be inserted, root entry of the subtree, its parent node and the pointer to stack S. It routes the entries to the next buffer nodes or leaf nodes using the original MVBT routing algorithm (see line 25). Routing through $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ levels of the subtree is done in memory. Note that we consider only live

index nodes for routing. In Lemma 2, we show that the live nodes of a subtree of height $\log_{\frac{B}{4}} \frac{M}{16B}$ always fit in memory.

We also update weight information of the nodes in line 16 of Algorithm 3. The weight information is recorded in the corresponding index entries. Moreover, buffer nodes are pushed into S if the size of their buffer is greater than $\frac{M}{4}$ (see lines 17–18).

Before we route a record one level down, the weight conditions of the node are checked (see line 2, Algorithm 3). If the conditions are violated, node reorganization is triggered (merge, time-split, key-split or merge-key-split). For buffer nodes, we empty their buffers first, since we need to enforce an overlap of the safe intervals. This ensures that there is no entry in the buffer with a time stamp smaller than that of the entry to be inserted. Otherwise, some records could lie outside the lower interval boundaries.

Thereafter, the node is reorganized in a top-down fashion (see call *SplitNode*, line 9, Algorithm 3). For the routing, only the live part of a tree is considered. Logically deleted nodes are released from memory. Index entries of the newly created nodes are posted to the parent node (see function *ExpandParent*). It is crucial for our algorithm that *this function does not trigger recursive parent splits towards the root node* due to weight balancing. If we perform a key-split or a merge-key-split, we adjust the node pointer according to the key of the entry (line 10, Algorithm 3). The root node split is handled in line 12 of Algorithm 3 similar to Algorithm 1 lines 10–15.

The procedure *ClearBuffer* (see Algorithm 4) is executed if either one of the following cases occur:

1. The buffer has more than $M/4$ records.
2. The buffer node has to be reorganized (e.g. due to weight violation).
3. The procedure *ClearAllBuffers* is executed (see line 8, Algorithm 2, lines 4–8 Algorithm 3).

In case 2. or 3., buffers are emptied completely. We push records in two batches of maximal size $M/4$. After processing each batch, we clear all child buffers with more than $M/4$ (see line 8 and 13, Algorithm 4). In addition, buffers belonging to the lowest buffer nodes are also completely emptied (even if they contain more than $M/4$ records). In all other cases, only the first $M/4$ records of a buffer are pushed down to the next buffer node. By this, we ensure that no buffer will have more than $M/2$ records and we avoid cascading buffer emptying processes [27].

Algorithm 4: ClearBuffer

```
   Input: Index Entry idx
1  buffer ← load buffer of idx ;
2  v ← load the node of idx ;
3  initialize S;
4  for i = 1 to min(M/4, buffer size) do
5      e ← Dequeue (buffer) ;
6      cRoot ← ChooseSubTree (v, e ) ;
7      PushDownEntry (e, cRoot, v, S, ts(e)) ;
8  foreach i in S do ClearBuffer (i) ;
9  if idx has weight violation or level == ⌊log_{B/4} M/16B⌋ or ClearAllBuffers then
10     foreach e in buffer do
11         cRoot ← ChooseSubTree (v, e) ;
12         PushDownEntry (e, cRoot, v, S, ts(e)) ;
13     foreach i in S do ClearBuffer (i) ;
```

4.3.2 Weight Balancing

Recall that we use two weight counters to track the temporal progress of a node v. Live weight $w(v)$ is used for live records. $t(v)$ tracks the number of update and insert records pushed to a node v (or to its buffer) on level l since its creation. We need this weight counter to properly estimate the safe interval as execution of the *update* operation (logical delete followed by insert) on a leaf node does not change the number of live entries.

The original index entries of MVBT are extended with two weight counters, w and t. MVBT$^+$ maintains w and t for each live node in its associated live index entry. Both counters are updated in a top-down fashion (see Algorithm 3 line 21). Every time a record is passed one level down, we update the node weight information as follows:

$$w(v) = \begin{cases} w(v)+1 & \text{if insert} \\ w(v)-1 & \text{if delete} \\ w(v) & \text{if update} \end{cases}$$

$$t(v) = \begin{cases} t(v)+1 & \text{if insert} \\ t(v) & \text{if delete} \\ t(v)+1 & \text{if update} \end{cases}$$

Recall that the original MVBT triggers time splits and further reorganizations like merges or key-splits only with respect to the number of physical entries. This occurs if a

Setting	Description
$a = \frac{B}{4}$	branching factor
$a^l \cdot \frac{B}{4} \leq w(v) \leq a^l \cdot B$	valid live weight
$w(t) \leq t(v) \leq a^l \cdot B$	valid operation weight
$a^l \cdot \frac{3B}{8} \leq w(v) \leq t(v) \leq a^l \cdot \frac{7B}{8}$	strong weight condition
$i \cdot \left\lfloor \log_{\frac{B}{4}} \frac{M}{16B} \right\rfloor$, i= 1, ...	buffer levels
$\frac{M}{2}$	maximal buffer capacity

Table 4.2: MVBT$^+$ settings

node has not enough live entries or the physical bound is achieved (see Section 2.2). We carry over the same idea to weight balancing. Thus, we limit live and operation weights $w(v)$ and $t(v)$ (length of safe interval) for each node. The allowed ranges exponentially increase with each level. The ratio of the minimal and the maximal value of $w(v)$ and $t(v)$ for level l is a constant.

We adapt weight balancing as follows: the branching factor a is set to $a = \frac{B}{4}$. The valid live weight of node v on level l is between:

$$\textit{live-condition: } a^l \cdot \frac{B}{4} \leq w(v) \leq a^l \cdot B \tag{4.1}$$

The valid operation weight $t(v)$ of the node v on level l is between:

$$\textit{operation-condition: } w(v) \leq t(v) \leq a^l \cdot B \tag{4.2}$$

Immediately after node reorganization, the live weight and operation weight of the newly created node should fulfill the following *Strong-weight-condition:* (see Table 4.2):

$$a^l \cdot \frac{3B}{8} \leq w(v) \leq t(v) \leq a^l \cdot \frac{7B}{8} \tag{4.3}$$

Node reorganization is triggered in two cases: if either

$$w(v) \leq a^l \cdot \frac{B}{4}$$

or

$$t(v) \geq a^l \cdot B$$

These two conditions are checked in line 2 of Algorithm 3. In line 9 of the algorithm function *SplitNode* is called. It runs the node reorganization and logically deletes the current index entry in the parent node. Each reorganization of node v on level l starts

w	t	w_n neighbor live weight	Operation
$a^l(\frac{3B}{8}) < w < a^l(\frac{7B}{8})$	$t \geq a^l B$		*time-split*
$w \geq a^l(\frac{7B}{8})$	$t \geq a^l B$		*key-split*
$w \leq a^l(\frac{3B}{8})$	$t \geq a^l B$	$w_n + w < a^l(\frac{7B}{8})$	*merge*
$w \leq a^l(\frac{3B}{8})$	$t \geq a^l B$	$w_n + w \geq a^l(\frac{7B}{8})$	*merge-key-split*
$w \leq a^l(\frac{B}{4})$	$t < a^l B$	$w_n + w < a^l(\frac{7B}{8})$	*merge*
$w \leq a^l(\frac{B}{4})$	$t < a^l B$	$w_n + w \geq a^l(\frac{7B}{8})$	*merge-key-split*

Table 4.3: MVBT$^+$ node reorganization conditions

with a time split. It creates a new node v_t and copies live entries into v_t.

The new live weight $w(v_t)$ and operation weight $t(v_t)$ are set to value of $w(v)$. If $w(v_t)$ violates *strong-weight-condition* (see Condition 4.3) one of the following operations are executed: merge, key-split and merge-key-split (see Table 4.3). In the case of a merge, the live weight w_n of siblings have to be considered to choose the type of reorganization (see Table 4.3). We release the dead node from memory and use the free space for a new live node. In the case of a merge operation we update $w(v_t)$ and $t(v_t)$ as follows $w(v_t) \leftarrow w(v_t) + w_n$ and $t(v_t) \leftarrow t(v_t) + w_n$.

The original key-split and merge-key-split of MVBT are also adapted to weight balancing. Key-splits on leaf nodes are identical to MVBT key-splits. This is different for index nodes. For these nodes, we iterate over the sequence of index entries and sum up their weights until the sum is at least equal to $(\frac{1}{2} \cdot a^l \cdot \frac{7B}{8})$ ($\frac{1}{2}$ of the upper bound of *strong-weight-condition*). Then, we split at the child entry where the iteration stopped. Afterwards, we adjust the weight counters of the created nodes. Since the maximal weight of this child is limited by $a^{l-1}B$, we find an almost balanced split. The branching factor a should be greater than 16. This follows from the following inequality [28, 35]:

$$\frac{1}{2} \cdot (a^l \frac{7B}{8}) - a^{l-1}B \geq a^l \frac{3B}{8} \tag{4.4}$$

Hence, this inequality ensures that a balanced split can be always found.

The lower bound of the live weight is $a^l \frac{B}{4}$. This ensures that after merging, the live weight of a new node is always greater than the lower bound of *strong-weight-condition* (since $2 \cdot a^l \frac{B}{4} \geq a^l \frac{3B}{8}$). The weight constraints and our reorganization operations guarantee that a minimum of $\frac{a^l B}{8}$ operations (inserts, deletes, updates) have to occur before node v on level l is reorganized again.

In line 15 of Algorithm 3, the newly created index entries are posted to the parent node (function call *ExpandParent*). Recall again that the insertions of these new entries do not cause a split of the parent node. Let us sum important properties in the following

lemata; their proofs can be found in the appendix.

Lemma 1. *For $a = \frac{B}{4}$, the number of entries in a MVBT$^+$ node is at most $6B = \Theta(B)$.*

Proof of Lemma 1: For $l = 0$ we have the same settings as MVBT leaves. Let v be node on level $l > 0$. To proof the lemma we compute the maximal number of entries that can be pushed before next reorganization. Since the minimal live weight of the node is $a^l \cdot \frac{3B}{8}$ (see equations 4.3) we can perform up to $a^l \cdot \frac{5B}{8}$ inserts (*Operation-condition*) and $a^l \cdot \frac{6B}{8}$ deletes (*Live-condition*) on this node. In total we can push up to $a^l \cdot \frac{11B}{8}$ operations until $t(v)$ overflows or the minimal bound of $w(v)$ is achieved. According to the weight constraints, a child node of the node v is reorganized after at least $a^{l-1} \cdot \frac{B}{8}$ operations. After each child split we produce 2 new entries in worst. Thus, after $a^l \cdot \frac{11B}{8}$ operations we create: $\frac{2 \cdot a^l \cdot \frac{11B}{8}}{a^{l-1} \cdot \frac{B}{8}} = 22a$ new entries. The maximal number of entries stored in the node with weight $a^l \cdot \frac{3B}{8}$ before inserting $a^l \cdot \frac{11B}{8}$ is $\frac{a^l \cdot \frac{3B}{8}}{a^{l-1}\frac{B}{4}} = \frac{3a}{2}$. Thus, total number of entries stored in the node is $22a + \frac{3a}{2} \leq 24a = 6B$ for $a = \frac{B}{4}$. □

According to Lemma 1, index nodes can occupy up to 6 pages in worst case. These pages are simply organized as a linked list. The function *ExpandParent* lazily appends new pages to the index node in at most constant time. At first glance, this seems to affect the practical performance of the loading algorithm. However, we did not observe this worst case of 6 pages in our experiments. In only one of our experiments we observed a list of two pages. In all other cases, the index node corresponds to one physical page.

Lemma 2. *The number of live entries in the node on level $l > 0$ of MVBT$^+$ is between $\frac{B}{16}$ and B. The number of live nodes in a subtree of height $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ is bounded by $\frac{M}{2B}$.*

Proof of Lemma 2: (the proof is similar to weight property [28]) We consider node v on level l. The maximal live weight is $a^l B$. The minimal weight of the live child is $a^{l-1}\frac{B}{4}$. Thus, the maximal number of live entries is $a^l B / a^{l-1}\frac{B}{4} = 4a$. Since $a = \frac{B}{4}$ we get the bound. The minimal live weight of the node is $a^l \frac{B}{4}$ (also operation weight $t(v)$, since $w(t) \leq t(w)$). The maximal live weight of the child is $a^{l-1}B$. Thus, the minimal number of entries is $a^l \frac{B}{4} / a^{l-1}B = a/4 = B/16$. We consider a sub-tree T of height $i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ on level i. Without loss of generality, we assume that $\log_{B/4} \frac{M}{16B}$ is an integer value. The maximal live weight of T is $a^{i \cdot \log_{B/4} \frac{M}{16B}} \cdot B$. Since the minimal weight of the tree on level $(i-1) \cdot \log_{B/4} \frac{M}{16B}$ is $a^{(i-1) \cdot \log_{B/4} \frac{M}{16B}} \cdot B/4$. Using the same argument as above, we obtain maximal number of buffer nodes $\frac{M}{4B}$, referenced by T. Thus, the overall number of live nodes is bounded by $2 \cdot \frac{M}{4B}$. □

Lemma 2 ensures that the live part of a subtree fits always in memory, since for the routing only live nodes are considered. This explains also the choice of height $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ for the buffer-tree configuration.

Before a node reorganization is executed, we check if the node has a buffer (see lines 4–8 in Algorithm 3). The buffer of the buffer nodes is emptied if the weight condition of the node is violated. The maximum number of entries in the buffer is limited to $\frac{M}{2}$. We call the *ClearBuffer* function and if the node should be merged we call also this function for a neighbor node. Buffers are always emptied before node reorganization to enforce an overlap of safe intervals (we synchronize nodes according to the time dimension). This operation must not happen frequently, since stopping the process and buffer emptying have worst case $O(M/B)$ I/O costs per buffer level. At least after each $\Theta(M)$ operation, a weight violation on the buffer nodes occurs.

Lemma 3. *MVBT$^+$ nodes on level $i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ with $i = 1, \ldots$ are reorganized again after at least $\Theta(M)$ operations (insertions, deletions, updates).*

Proof of Lemma 3: We consider the lowest buffer node v. The lowest level is $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$. According to weight conditions the minimal number of entries needed for next reorganization is $a^l \frac{B}{8}$. Thus, the node v is reorganized after $a^{\log_{B/4} \frac{M}{16B}} \frac{B}{8}$ operations. Since $a = \frac{B}{4}$ we get $\frac{M}{16B} \cdot \frac{B}{8} = \frac{M}{128} = \Theta(M)$. □

After the buffer is emptied completely, we continue with node reorganization. Finally, we assign a new buffer to a live node and drop buffers of temporal predecessors.

MVBT$^+$ has the same asymptotic I/O bounds on update and space as a MVBT loaded by tuple-by-tuple method. The following lemma holds for MVBT$^+$.

Lemma 4. *The MVBT$^+$ height loaded with N records is $O(\log_B N)$ and its space is $O(N)$.*

Proof of Lemma 4: A sub tree with a root node on level l references $O(a^l \cdot B)$ live elements in the live leaf nodes. After insertion of N operation entries at most N entries are alive. The minimal live weight of the live root node is at least $N \geq 2 \cdot a^{l-1} \cdot \frac{B}{4}$. Since $a = \frac{B}{4}$ level of the root node is $O(\log_B N)$.

A leaf node (level = 0) is reorganized at least after performing $\frac{B}{8}$ operations. In worst we create two new leaf nodes. Node v node on level l is reorganized after at least $a^l \frac{B}{8}$ operations. In worst we create also two new nodes. Thus, after N operations we create up to: $2 \cdot N \sum_{l=0}^{\log_a N} \frac{1}{\frac{B}{8} \cdot a^l} \leq 16 \cdot N/B \sum_{l=0}^{\infty} \frac{1}{a^l}$ nodes. Since $a \geq 16$ and the node capacity is $O(B)$ according to lemma 1 we obtain the result of $O(N)$ space. □

4.3.3 Runtime

In this section, we outline the proof of the following theorem:

Theorem 4.1. *The cost for loading N records in an initially empty MVBT$^+$ with branching parameter $a = B/4 \geq 16$ is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os.*

First, we consider the costs of emptying buffer of node v on level $l = i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ and the node reorganization. Thereafter, we discuss the emptying of all buffers after insertions of N records (see function *ClearAllBuffers* in Algorithm 2). Records are pushed down in batches of size $\frac{M}{4}$ towards leaves starting from the root node. We consider two cases of buffer emptying, caused either by buffer overflow or by the node weight violation.

Buffer-Overflow: I/O costs of loading a subtree of height $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ in memory is $O(M/2B)$ I/Os (see Lemma 2). The overall split costs are limited by $O(M/B)$ I/O between two buffer levels, since the weight of the subtree root is not violated (see Lemma 2). Thus, $\frac{M}{4}$ entries routed one level down causing $O(\frac{M}{B})$ I/Os. The I/O cost per entry is $O(\frac{1}{B})$. Entries pass $O(\log_B N/B)$ nodes before they are inserted in the leaf nodes. Yet, we pay I/Os only on each $l = i \left\lfloor \cdot \log_{B/4} \frac{M}{16B} \right\rfloor$ level. Thus, the overall I/O cost per entry is $O(\frac{1}{B} \cdot \frac{\log_{B/4} N_i/B}{log_{\frac{B}{4}} \frac{M}{16B}}) = O(\frac{1}{B} \cdot \log_{\frac{M}{B}} N/B)$.

Weight-Violation: Reorganization of a buffer node stops the buffer emptying process and triggers up to two buffer emptying processes. In the case of merge or merge-key-split we also empty the buffer of the sibling node. In the worst case, we write dirty subtree nodes to the disk using at most $O(\frac{M}{2B})$ I/Os (see Lemma 2). The buffers of both nodes contains up to $\frac{M}{2}$ entries each, since we push data in portions of $\frac{M}{4}$ records. For emptying both buffers we pay up to $O(\frac{M}{B})$ I/Os. Total I/O costs are bounded by $O(\frac{M}{B})$. In worst case, we pay $O(M/B)$ I/O for buffer node reorganization after each $\Theta(M)$ operations (see Lemma 3). The total worst case costs of lower buffer nodes splits are $O(N/B)$. The costs for all remain buffer levels are:

$$N \cdot \sum_{i=1}^{\log_{\frac{M}{16B}} N/B} \frac{1}{(\frac{B}{4})^{i \cdot \log_{B/4} M/16B} \cdot \frac{B}{8}} \cdot O(\frac{M}{B}) = O(\frac{N}{B} \log_{M/B} \frac{N}{B})$$

Thus, after insertion of N entries in an empty MVBT$^+$ we pay $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ for emptying full buffers and buffer node reorganizations.

Finally, we show that the emptying of all buffers after insertion of N operation entries is bounded by $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os. As shown above, costs for emptying full buffers

and for emptying due to weight violation is bounded by $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os. The costs of emptying the remaining non-full buffers is bounded by $O(N/B)$. Since the lowest buffer node level is $l = \left\lfloor \log_{\frac{B}{4}} M/16B \right\rfloor$, the number of buffer nodes after insertion of N operation entries is limited by $O(\frac{N}{B}/(M/B))$. The worst-case-cost of the buffer emptying process is $O(M/B)$. Therefore, on each $O(\frac{N}{B}/(M/B))$ we pay $O(M/B)$ I/O resulting the $O(N/B)$ bound. Combining this result with buffer emptying caused by buffer overflow and weight violations before *ClearAllBuffers* yields the desired asymptotic bounds.

4.4 Bulk Update

In this section, we briefly describe how to insert a sequence of records efficiently into a non-empty MVBT$^+$ whose buffers are entirely empty. To implement a bulk update, we follow the ideas presented in [27] with a minor modification of algorithm 2: we use the current live root and its buffer instead of a pointer to an empty leaf node. Bulk update appends records to a current root buffer and if applicable pushes entries towards leaf nodes. We call function *ClearAllBuffers* by the end of procedure. Since we load records into existing MVBT$^+$, the records are routed only through the live nodes. By this, we obtain the following I/O cost for a bulk update:

Theorem 4.2. *The cost for a bulk update of N records on an existing MVBT$^+$ with N' live records and empty buffers is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N+N'}{B} + \frac{N'}{B})$ I/Os.*

Analogous to the proof of Theorem 4.3, we obtain the I/O bound.

4.5 Practical Considerations

We designed memory and weight-balancing settings from a worst case perspective. With memory $M \geq 4B^2$ we have buffers at least on each index level. However, according to our experimental results less memory is sufficient to achieve the desired performance. In practical applications we assume that at least $M \geq B^2$ memory is available. Therefore, we set buffers on each $i \cdot \max\left\{\left\lfloor \log_{B/4}(M/16B)\right\rfloor, 1\right\}$ level.

Note that the introduced weight-balancing can be used without the buffer tree technique. According to Lemmas 2 and 4 a weight-balanced MVBT has the same asymptotic bounds on space, query and update time as the MVBT. However, update and query operations have a constant overhead. Since in general we store the weights in index entries, on update we need always to write back the index nodes to a disk. In contrast, this overhead is amortized over a batch of update operations while bulk-loading.

Nevertheless, MVBT$^+$ can be used for normal updates using $O(\log_B N)$ I/Os per update. Consequently, we need to manage weight information in the nodes, similar to a bulk loading case. Yet, we do not attach buffers for a single update operation. To insert a single update record, we slightly modify the bulk update algorithm. We do not attach buffers to the live nodes and do not call *ClearBuffer* function. The modified Algorithm 3 pushes an update record down to the leaf level, while updating node weight information and if applicable performing node reorganizations.

In the case of the bulk update, buffers are attached to an existing by MVBT$^+$ with very small overhead, as all information such as the weight and the level of a node is available. Thus, we can lazily attach buffers during the bulk update. The live weight of the live root displays the number of live records stored in a tree. Since the worst case cost of a bulk update also depends from the number of currently stored live records, we can directly derive the upper I/O bound of a bulk update from the live weight information. If the size of a batch N is known in advance, we have two options:

1. The first option is: inserting records without attaching the buffer yields worst case cost $N \log_B(N + N')$ I/Os (This can be speed up by using e.g. LRU-Buffer).

2. The second option is an executing the bulk update procedure. We attach buffers on demand and subsequently empty all buffers.

We can choose the update strategy based on the estimated worst case cost. This hybrid strategy yields $\min\{N \log_B(N + N'), \frac{N}{B} \log_{\frac{M}{B}} \frac{N+N'}{B} + \frac{N'}{B}\}$ I/O cost.

4.6 A Class of MVBT$^+$Trees

In this section we present a class T_m of multiversion B-trees. T_m has worst case I/O bounds for bulk loading and bulk update as well as for space, update and query time.

We define $T \in T_m$ as a MVBT$^+$ from a class of T_m using the following parameters B, d, ϵ, a: B is page capacity. $d \in \Theta(B)$ is the minimal number of live records per page. ϵ is defined as a fraction of parameter d such that $0 < \epsilon \leq 1$ and $d \cdot \epsilon$ operations is needed to trigger next leaf node reorganization. a is a branching factor of internal nodes. Table 4.4 summarizes notations used in this section.

Further, we assume T_m uses Algorithms 2, 4 and 3 (see Section 4.3.1) for bulk loading and bulk update, respectively. It uses also a modified insert algorithm for a single update.

T belongs to T_m if the following conditions are fulfilled:

Setting	Description
$d \in \Theta(B)$	minimal number of live entries per leaf node
$a \in \Theta(B)$	branching factor of internal nodes
$0 < \epsilon \leq 1$	fraction of d

Table 4.4: MVBT$^+$ settings

1. *capacity-condition*: The value of branching parameter a and the minimal number of live records per leaf node $d = a$ are equal and bounded by $\Theta(B)$[1].

2. *tree-consistency-conditions*: Node v valid weights $w(v)$ and $t(v)$ fulfills

$$\textit{live-condition: } a^l \cdot d \leq w(v) \leq a^l \cdot B \tag{4.5}$$

and

$$\textit{operation-condition: } w(v) \leq t(v) \leq a^l \cdot B \tag{4.6}$$

Immediately after node reorganizations $w(v)$ and $t(v)$ fulfills *strong-weight-condition:*

$$a^l \cdot (d + d \cdot \epsilon) \leq w(v) \leq t(v) \leq a^l \cdot (B - d \cdot \epsilon) \tag{4.7}$$

Reorganizations are performed as described in Section 4.3.2. Table 4.5 summarizes operations triggered based on node weights.

3. *split-inequality*: Parameters a, d and ϵ satisfy the following inequality:

$$\frac{1}{2} \cdot (a^l(B - d \cdot \epsilon)) - a^{l-1}B \geq a^l(d + d \cdot \epsilon) \tag{4.8}$$

4. *merge-inequality*: Parameter a, d and ϵ satisfy the following inequality:

$$2 \cdot a^l d \geq a^l(d + d \cdot \epsilon) \tag{4.9}$$

5. *buffer-node-condition*: Buffers of size $M/2$ are assigned to nodes on level l

[1] We introduce both parameters, as d is inherited from MVBT and a is by weight-balancing technique. Both have different meanings, however, in order to achieve the desired I/O complexity we initialize them with the same value.

$$l = i \cdot \lfloor \log_a \frac{M \cdot d}{4 \cdot B^2} \rfloor \text{ for } i = 1, \ldots \tag{4.10}$$

Buffers are emptied as described in Section 4.3.1.

The I/O model enforces *capacity-condition*. Due to *tree-consistency-conditions* the next node reorganization on node v on level l occurs at least after $d \cdot \epsilon \cdot a^l$ operations (insert, delete and update records) since it creation. In order to guarantee that a balanced split of internal nodes can be computed, parameters d, ϵ, a must fulfill *split-inequality*, as MVBT$^+$ key-split splits internal nodes based on the node weights instead of number of entries. Further, parameters d, ϵ, a must fulfill *merge-inequality*, since after the merge the new node live weight should be at least $a^l(d + d \cdot \epsilon)$.

The update records (operations) are routed through the set of live nodes. According to the *buffer-node-condition* we have $M/2$ memory space to hold this live node set. The maximal number of live entries of node v on level l is given by $\frac{a^l B}{a^{l-1} d}$. As we initialize branching parameter a and parameter d with the same value, the number of live entries is equal to page capacity B. By this, the live child number of internal node is also bounded by B. To this end, the number of live entries per internal nodes should be bounded by B.

Since each live node stores up to B live entries, the maximal number of live nodes loaded for routing in main memory is bounded by $\frac{M}{2B}$. Attaching buffers on levels $l = i \cdot \lfloor \log_a \frac{M \cdot d}{4 \cdot B^2} \rfloor$ for $i = 1, \ldots$ allows us to load up to $\frac{M}{2B}$ live nodes. This follows from the following lemma:

Lemma 5. *The number of live nodes of a sub tree with height* $\lfloor \log_a \frac{M \cdot d}{4 \cdot B^2} \rfloor$ *is bounded by* $M/2B$

Proof. Without loss of generality, the value $\log_a \frac{M \cdot d}{4 \cdot B^2}$ is an integer. Consider sub tree root node v on level $i \cdot \log_a \frac{M \cdot d}{4 \cdot B^2}$. Let v be a buffer node. Let v_d be a descendant buffer node on the next buffer level level $(i-1) \cdot \log_a \frac{M \cdot d}{4 \cdot B^2}$. The maximal live weight $w(v)$ is $a^{(i \cdot \log_a \frac{M \cdot d}{4 \cdot B^2})} \cdot B$. The minimal live weight of a node on level $(i-1) \cdot \log_a \frac{M \cdot d}{4 \cdot B^2}$ is $a^{(i-1) \cdot \log_a \frac{M \cdot d}{4 \cdot B^2}} \cdot d$. The maximal number of live nodes on level $(i-1) \cdot \log_a \frac{M \cdot d}{4 \cdot B^2}$ is given by $a^{(i \cdot \log_a \frac{M \cdot d}{4 \cdot B^2})} \cdot B / a^{(i-1) \cdot \log_a \frac{M \cdot d}{4 \cdot B^2})} \cdot d = M/4B$. Hence, the overall number of live nodes is bounded by $M/2B$. □

Remain $M/2$ memory is used for buffers. Buffers are emptied in batches of size $M/4$ as described in Section 4.3.1. In Section 4.3.1 we parameterized d, ϵ and parameter a with $d = \frac{B}{4}$, $\epsilon = \frac{1}{2}$ and $a = \frac{B}{4}$. These parameters fulfill T_m conditions.

w	t	w_n neighbor live weight	Operation
$a^l(d \cdot (1+\epsilon)) < w < a^l(B - d \cdot \epsilon)$	$t \geq a^l B$		*time-split*
$w \geq a^l(B - d \cdot \epsilon)$	$t \geq a^l B$		*key-split*
$w \leq a^l(d \cdot (1+\epsilon))$	$t \geq a^l B$	$w_n + w < a^l(B - d \cdot \epsilon)$	*merge*
$w \leq a^l(d \cdot (1+\epsilon))$	$t \geq a^l B$	$w_n + w \geq a^l(B - d \cdot \epsilon)$	*merge-key-split*
$w \leq a^l d$	$t < a^l B$	$w_n + w < a^l(B - d \cdot \epsilon)$	*merge*
$w \leq a^l d$	$t < a^l B$	$w_n + w \geq a^l(B - d \cdot \epsilon)$	*merge-key-split*

Table 4.5: MVBT$^+$ node reorganization conditions

Theorem 4.3. *$T \in T_m$ is asymptotically optimal multiversion B-tree (linear space, logarithmic update and query time). The I/O cost for the bulk loading T is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ and the bulk update $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N+N'}{B} + \frac{N'}{B})$ respectively.*

Proof. Firstly, we obtain space and height bounds in manner similar to the proof of Lemma 4. The height of T_m trees is bounded by $O(\log_B N)$, since the value of branching parameter a is equal to minimal live record capacity d and bounded by $\Theta(B)$. Due to the fact that $d \cdot \epsilon$ parameter is bounded by $\Theta(B)$, we obtain the space bound.

Secondly, the I/O complexity of a single update operation is bounded by $O(\log_B N)$. The algorithm computes a path from a live root to a live node. As the maximal number of entries of internal node is bounded by $\Theta(B)$, update procedure always loads up to $O(1)$ nodes per level and writes them back due to the node weights. In order to show that internal node contains maximal up to $\Theta(B)$ entries, we use the same arguments as in Lemma 1: the node minimal live weight is $a^l(d + d \cdot \epsilon)$, by this, we can perform up to $a^l(B - ((d + d \cdot \epsilon)))$ inserts and $a^l(B - d)$ deletes. In total we can push up to $a^l(2B - 2d - d \cdot \epsilon)$ operations until $t(v)$ overflows or the minimal bound of $w(v)$ is achieved. According to the weight constraints, a child node of the node v is reorganized after at least $a^{l-1}(d \cdot \epsilon)$ operations. After each child split we produce 2 new entries in worst. Thus, after $a^l(2B - 2d - d \cdot \epsilon)$ operations we create: $\frac{2 \cdot a^l(2B-2d-d\cdot\epsilon)}{a^{l-1}(d\cdot\epsilon)} = \frac{4B-4d-d\cdot\epsilon}{\epsilon}$ new entries as $d = a$. The maximal number of entries stored in the node with weight $a^l(d + d \cdot \epsilon)$ before inserting $a^l(2B - 2d - d \cdot \epsilon)$ is $\frac{a^l(d+d\cdot\epsilon)}{a^{l-1}d} = d + d \cdot \epsilon$. Thus, total number of entries stored in the node is $(\frac{4B-4d-d\cdot\epsilon}{\epsilon} + d + d \cdot \epsilon) \in \Theta(B)$ for $a = d \in \Theta(B)$ and $d \cdot \epsilon \in \Theta(B)$. The I/O complexity of the node reorganization per level is bounded by $O(1)$ (there are up to four nodes involved, node to be reorganized, its neighbor and two new created live nodes). Therefore, since the overall I/O time per level is $O(1)$, the single update on T_m is performed in $O(\log_B N)$ I/O time.

Thirdly, we show the bounds for the bulk loading and the bulk update. The lowest buffer node v level is $\lfloor \log_a \frac{M \cdot d}{4 \cdot B^2} \rfloor$. Without loss of generality, we consider this value

$\log_a \frac{M \cdot d}{4 \cdot B^2}$ to be integer. Then, the minimal number of operations needed to trigger next node reorganization for node v after its creation is equal to $a^{\log_a \frac{M \cdot d}{4 \cdot B^2}} \cdot d \cdot \epsilon = \frac{M \cdot d^2 \cdot \epsilon}{4 \cdot B^2} \in \Theta(M)$. Combining this result with arguments from Theorem 4.3 we obtain the I/O bounds. □

4.7 Experiments

In this section, we report the main results of our algorithms for bulk loading and bulk update. We discuss the MVBT$^+$ and MVBT query performance and compare the results to those of a bulk-loaded R-trees.

4.7.1 Workload Generation

Workloads used in our experiments are designed similarly to other experimental studies with versioned databases [85, 86, 122]. We consider two different types of workloads: one for index loading and the other for queries.

Our loading workload consists of six files: $d50$, $u0$, $u25$, $u50$, $u75$, $u100$. Each of them contains $10'000'000$ operations[42][2]. For all data sets, the first $1'000'000$ operations are insertions (10% of the data set). The remaining 90% of the file consists of a mix of insertions, deletions and updates. Data sets are named after these specific operations. For example the file $d50$ consists of $1'000'000$ insertions followed by a mix of insertions ($4'500'000$) and deletions ($4'500'000$). File $u75$ consists of $1'000'000$ initial insertions followed by a mix of insertions ($2'250'000$) and updates ($6'750'000$). Inserted record keys are obtained from a permutation of $1, ..., k$, where k denotes the total number of insertions in the particular workload. Deletions and updates randomly select one from all live records. In the following, we use the term update to refer to all of these operations.

For each file from the loading workload we consider three query files qr_1, qr_2, qr_3. Each query file contains two-dimensional range queries (key range and time range) of the same absolute selectivity. qr_1 consist of $10'000$ queries with 100 answers. qr_2 consists of $1'000$ queries with $1'000$ answers and qr_3 contains $1'000$ queries with $10'000$ answers. Queries are uniformly distributed in the two-dimensional space.

4.7.2 Experimental Setup

All algorithms are implemented in Java using the XXL library[3]. MVBT$^+$ is implemented on top of the existing MVBT by associating two additional weight counters w and t to

[2] All files are available online http://dbs.mathematik.uni-marburg.de/Home/Downloads
[3] xxl.googlecode.com

	MVBT+		MVBT-LRU	
	B leaf	B index	B leaf	B index
4KB	97	97	97	121
8KB	197	197	197	245
16KB	397	397	397	493

Table 4.6: Node capacity

an index entry.

We conducted our experiments on a system running Windows 7 equipped with an Intel I7 CPU, 8GB of main memory, a magnetic disk (WD Caviar Black 1002FAEX, 1TB) and a SSD (Corsair Force 3 SSD, 120 GB). To avoid an impact of the operating system on our experiments, we used only the raw device interface.

We ran our experiments with pages of different sizes: 4KB, 8KB and 16KB. Table 4.6 reports the page capacities in the number of tuples. Each page contains header information that occupies 102 bytes (like the level, number of entries, and pointers to temporal predecessors). The size of a versioned record in a leaf occupies 41 bytes (17 bytes for the time interval, 8 bytes for the key and 16 bytes for the payload). The size of an MVBT index entry is 33 bytes (8 bytes for the node pointer, 8 bytes for the key, 17 bytes for a time interval), whereas the size of a MVBT$^+$ index entry is 41 bytes (because of the additional weight counters w and t).

The available main memory varied from 0.8 MB up to 16 MB. These numbers sound very small, however the size of the memory has only a marginal impact (base in the log-factor) on the loading performance. We assigned buffers to each $i \cdot \max\left\{\left\lfloor \log_{B/4}(M/16B)\right\rfloor, 1\right\}$ MVBT$^+$ level. Thus, one buffer is assigned to each internal live node. Buffer size varied from 200 KB up to 4MB because it equals to $\frac{1}{4}$ of the available memory.

Wall clock time and number of I/Os were employed to measure loading performance. The query performance was measured by the number of I/Os and the number of leaf accesses.

4.7.3 Bulk-Loading Results

In this section, we compare the performance of our new bulk loading method to iterative MVBT loading (update by update). MVBT-LRU and MVBT refer to iterative loading with and without a LRU-Buffer, respectively. MVBT-LRU and MVBT$^+$ always received an equal amount of memory.

Figure 4.3 reports the total number of I/Os required for loading MVBT$^+$ and MVBT for each workload file. We used a page size of 8 KB and a fixed memory size of 1.6 MB.

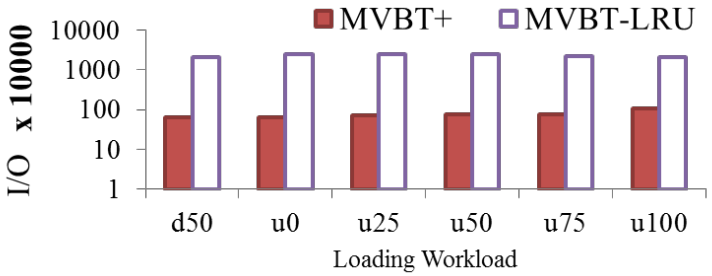


Figure 4.3: Loading performance (logarithmic scale) of MVBT-LRU and MVBT$^+$ (page size = 8 KB, memory size = 1.6 MB)

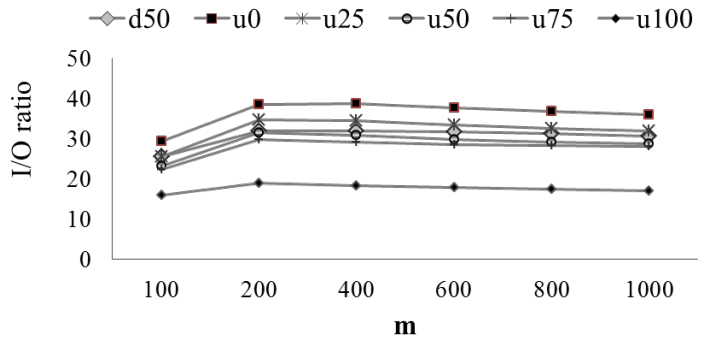


Figure 4.4: I/O Ratio of MVBT-LRU and MVBT$^+$ as a function of the memory size(page size = 8KB)

Results (number of I/Os) are given on a logarithmic scale. MVBT$^+$ clearly outperforms MVBT by a factor between 18 and 40.

Figure 4.4 depicts the ratio of I/Os required to load MVBT-LRU and MVBT$^+$ as a function of memory size. The best results for MVBT$^+$ are achieved for 200 pages. For larger memory sizes, I/O performance of MVBT-LRU improves slightly faster than the one of MVBT$^+$. The (relative) number of I/Os of MVBT$^+$ also increases with a growing number of updates. Due to a smaller number of live versions, the buffer of MVBT-LRU becomes more effective. For updates only (file $u100$), the LRU buffer contains all internal MVBT nodes, while a lot of reorganization steps are triggered for MVBT$^+$. Thus, the MVBT$^+$ performance improvements are only a factor of 15.

In Figure 4.5, the I/O ratio is depicted as a function of the page size for loading the data set $u50$. The memory capacity is set to 400 pages in total. The lower curve displays the ratio between I/Os required for loading MVBT-LRU and the ones required for loading MVBT$^+$. As expected from our theoretical results, we observe a linear improvement of the I/O performance with an increasing page size. For 16 KB pages, MVBT$^+$ runs faster

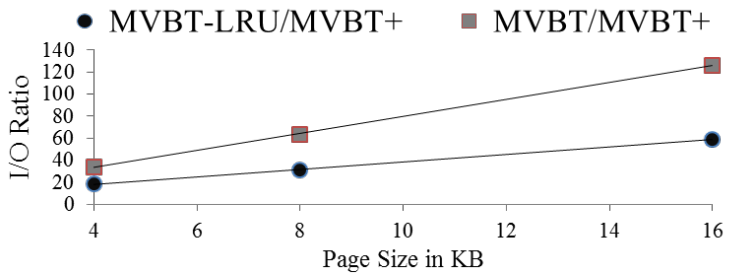


Figure 4.5: I/O ratio as a function of the page size ($u50$ data set, memory size $m = M/B = 400$)

than MVBT by a factor of 58. The upper curve illustrates the worst-case for MVBT when no LRU buffer is used. The curve shows the relative performance gains of MVBT in comparison to MVBT$^+$.

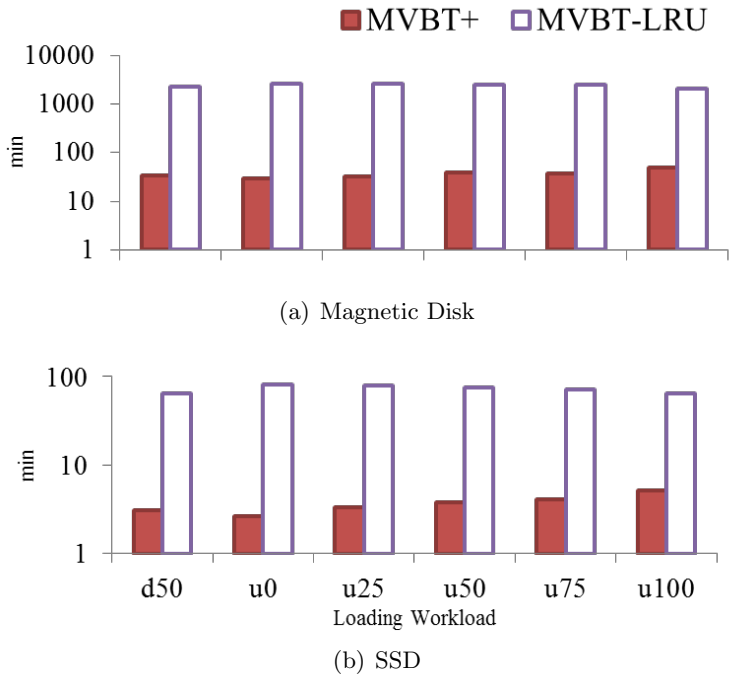


Figure 4.6: Loading times (logarithmic scale) of MVBT-LRU and MVBT$^+$ (page size = 8KB, memory size = 1.6 MB)

Figure 4.6 a and 4.6 b depict loading time in minutes, using either magnetic disk or SSD for all workload files. For the magnetic disk, loading MVBT$^+$ takes between 30 and 60 minutes, while MVBT-LRU requires between 30 and 40 hours. Greater performance

improvements (in comparison to the pure I/O numbers) are due to MVBT$^+$ uses a larger number of sequential I/Os than MVBT-LRU. For SSD, loading times of MVBT$^+$ are between 3 and 5 minutes, while MVBT-LRU requires between 64 and 82 minutes.

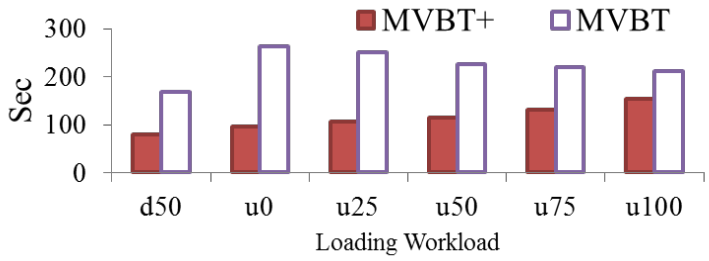


Figure 4.7: Loading times of MVBT-LRU and MVBT$^+$ in main memory

Figure 4.7 displays the wall clock time when the entire loading is performed in main memory. All nodes of the trees are kept in memory (without the need for serialization) and a LRU-buffer was not used anymore. The page capacity was set to 4 KB (B = 97). The MVBT+ still used buffers for all internal nodes and the buffer capacity was set to 100 pages. Though the entire loading runs in main memory, MVBT$^+$ still runs faster than iteratively calling the ordinary insertion algorithm of the MVBT by at least a factor of 1.37. The reason is that the higher locality of computation is not only important for disks, but also on the upper levels of the memory hierarchy.

Figure 4.9 depicts the average space utilization of index and leaf nodes. The leaf node utilization of MVBT$^+$ does not differ from the original MVBT. According to Lemma 1 the utilization of an index node is limited by $O(B)$. More precisely, an index node of MVBT$^+$ contains at most $6 \cdot B$ and at least $\frac{B}{16}$ entries. In our experiments, we observed that on average there are $B/2$ entries in one index node. This is less than for the original MVBT. We did not observe more than B per index node except very rarely for data set $u100$.

Figure 4.8 depicts the number of buffer node reorganization and the overall buffer emptying calls. As expected, with increasing number of update operations, there are more time splits than key-splits. Therefore, the number of operations needed to trigger node reorganization on level l is always roughly worst case $a^l \frac{B}{8}$. At least after each $a\frac{B}{8} = \frac{B^2}{32} = 1225$ entries the lowest buffer node should be reorganized (in our experimental setup). The number of the lowest buffer node reorganizations is limited by $10'000'000/1225 = 8164$. In all our experiments the overall number of buffer node reorganization was less than this number.

Table 4.7 shows space consumption of the resulting trees. The total space required

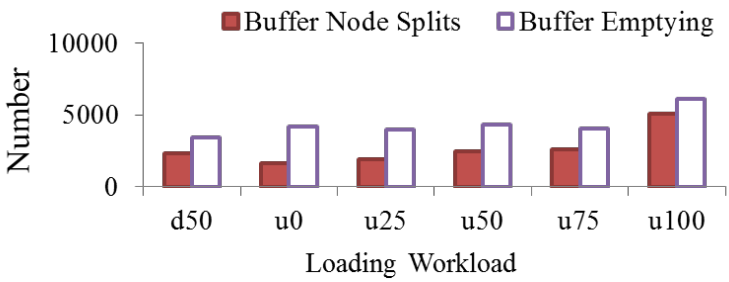


Figure 4.8: Average number of data pages needed to trigger buffer emptying process (8KB pages, M/B=200)

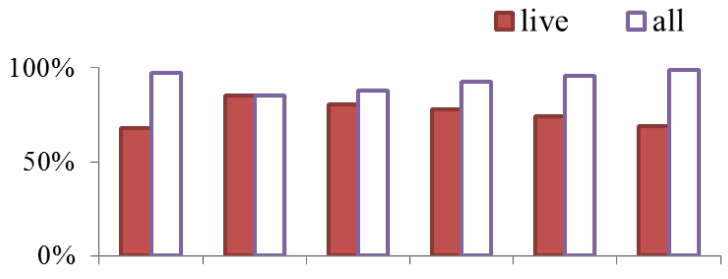
File	$d50$	$u0$	$u25$	$u50$	$u75$	$u100$
MVBT$^+$	629	1157	1241	1253	1261	1239
MVBT	605	1153	1215	1223	1235	1196

Table 4.7: Storage utilization of MVBT$^+$ and MVBT

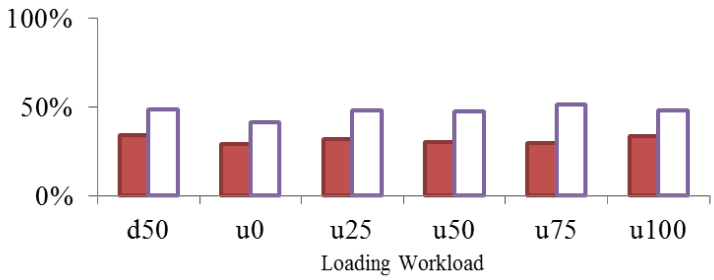
for MVBT$^+$ increases only slightly in comparison to MVBT. This is due to the larger index entries in which the weight counters w and t have to be kept. Moreover, weight balancing results in a lower storage utilization in the index nodes as shown in Figure 4.9 b.

4.7.4 Bulk Update Results

In addition to loading, we also conducted a series of experiments to measure the I/O efficiency of bulk updates on a given MVBT$^+$ and MVBT-LRU, respectively. For each data set we first executed 5′000′000 updates (50% of the total updates). Thereafter, we processed the remaining updates with a sequence of bulk updates (with a given batch size). Bulk updates on MVBT-LRU are again implemented by calling the update function one by one. Figure 4.10 depicts the I/O ratio of MVBT-LRU and MVBT$^+$ as a function of batch size. The memory size was set to 200 pages. MVBT-LRU required slightly less I/Os than MVBT$^+$ for batch sizes with less than 10′000 updates. The reason is that after the updates of the entire batch are performed many buffers contains only one or very few update operation. However, these buffers are forced to be emptied because this has to be performed after the batch. Note that these results are still in agreement with the asymptotically optimal worst-case bounds of Theorem 4.2. For batch sizes with more than 10′000 updates, the situation is different and MVBT$^+$ is still superior. For a batch size of 400K , for example, the MVBT$^+$ improvements over the MVBT-LRU are between 9 (for file $i0$) and 18 (for file $d50$).



(a) Leaf nodes storage utilization



(b) Index nodes storage utilization

Figure 4.9: Average storage utilization of leaf and index nodes (MVBT+, 8KB pages B=197=100%)

4.7.5 Query Workload Results

Workload file $u50$		qr_1	qr_2	qr_3
MVBT+	I/O	4.75	11.98	85.23
	leafs	1.78	8.88	80.62
MVBT-LRU	I/O	4.18	11.33	83.74
	leafs	1.79	8.89	80.8
R-TREE	I/O	116.3	124.92	197.34
	leafs	104.2	112.8	184.8

Table 4.8: I/Os and leaf accesses for query workload qr_1, qr_2, qr_3 and for MVBT+, MVBT and R-tree

We conducted a series of experiments running the query workloads on each data file. As expected, we observed almost the same number of leaf accesses for MVBT+ and MVBT. There are marginal differences, as merging of leaves might differ for MVBT and MVBT+. Due to different split strategies for index nodes, the original sibling of a leaf might belong to a different parent node in the case of MVBT+. The number of accesses to index nodes is higher for MVBT+ in comparison to MVBT (Table 4.8). The average

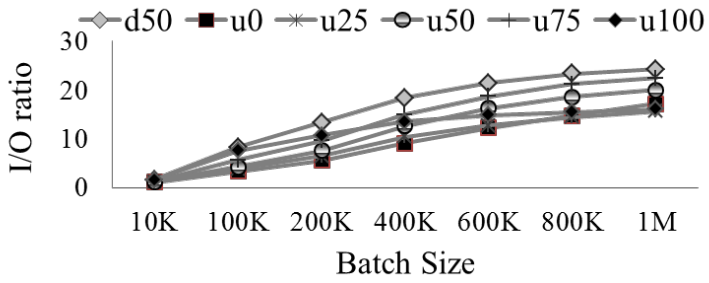


Figure 4.10: Bulk update, I/O Ratio MVBT-LRU / MVBT$^+$ 8 KB pages

number of I/Os per query are reported for the three query files on data file $u50$. However, only for small queries (qr_1) the increase in the number of I/Os for MVBT$^+$ over MVBT is close to 10%.

Additionally, we also report the query performance of an R-tree in Table 4.8. We built the R-tree using STR bulk loading algorithm [80], a popular loading method that is also utilized in commercial systems. Our results clearly show that the R-tree query performance is inferior to the MVBT performance. The reason is simply the high overlap among the nodes of the R-tree. This is particularly noticeable for small queries.

4.8 Conclusions

In this chapter we presented MVBT$^+$, which is the first partially persistent B$^+$tree that supports bulk loading in an asymptotically optimal number of I/Os and maintains all worst-case performance guarantees of the multiversion B-tree (MVBT). The results of our experimental studies showed that excellent loading times can also be achieved for various storage devices (magnetic disks, SSD, main memory). In comparison to previous loading approaches, i.e., loading by iterative updates, MVBT$^+$ loading is substantial faster by a factor linear to the page capacity. As MVBT$^+$ uses a weight balancing technique, fill degree of non-leaf nodes is slightly lower than for the original MVBT, but this leads to only a slight deterioration of the MVBT$^+$ query performance.

5 Query Adaptive Loading of R-trees

In this work we investigate the problem of efficient R-tree bulk loading techniques. The novelty of our work is that we consider a query profile for loading R-trees, since current loading approaches for R-trees disregard this. The query profile consists of a set

(a) Input set (b) Query set

Figure 5.1: Figure depicts minimal bounding rectangles of California streets data set and a set of query ranges Q

of query rectangles. Figures 5.1(a) and 5.1(b) depict set of minimal bounding rectangles (MBRs) of California streets with a query set (profile)[1]. Based on the statistical information about queries such as average size and their shapes, we are able to build R-tree minimizing the costs according to a widely used cost model for R-tree query performance [73, 115, 95]. Knowledge of a query profile influence the structure or R-tree nodes e.g. firstly, average query size influence the average capacity of R-tree nodes. Secondly, consider extreme example depicted in Figure 5.2. The query set for California streets with a high aspect ratio (ratio of side lengths) is plotted in Figure 5.2(a). Figure 5.2(b) displays leaf MBRs of R*-tree. The R*-tree disregards query profile and produces square shaped MBRs. Figure 5.2(c) depicts leaf MBRs of R-tree build using our query adaptive approach. It is beneficial to adapt the shape of leaf MBR to the average query shape.

[1]TIGER Data: http://www.census.gov/geo/www/tiger/

(a) Query set (b) Leaf MBRs R*tree (c) Leaf MBRs R-tree

Figure 5.2: Impact of query shape

Our research goal was to design a loading approach that fulfills the following requirements:

1. The algorithm should consider statistical information obtained from a query set to build R-trees that minimizes a cost model proposed by [73, 115, 95]. If there no query profile available, algorithm should minimize area of minimal bounding rectangles of the R-tree nodes.

2. The I/O costs of the algorithm should be bounded by the costs of external sorting.

3. The algorithms should be conceptual simple.

4. Devising a parallel and distributed version of the algorithms should be possible.

Regardless of the query profile knowledge our developed loading strategy substantially improves average query performance. Yet, if query profile is available our approach yield more better R-trees. The second requirement is motivated by the general trade-off between query performance and the R-tree construction time. Moreover, according to the results in [7] the problem of construction R-tree that minimizes the sum of node MBR areas is NP-hard. In order to fulfill second requirement, we reduce the complexity of multidimensional sets by sorting according to a space filling curve(SFC). This allows us to apply efficient dynamic programming scheme to find an optimal solution for the one-dimensional problem. As external sorting is an integral part of databases systems, we believe that the implementation and integration costs of our loading approach are low. The fourth requirement is motivated by recent volume growth of a processing data. Our bulk loading approach allows us developing efficient parallel and distributed version e.g. [97, 48, 9], as it based on sorting data according to SFC.

Major parts of this chapter are based on the following publications [8, 7, 4]. The chapter is organized as follows: In Section 5.1 we review the cost model used for optimization of R-trees. In Section 5.2 we present our loading framework. We discuss the crucial steps of our loading approach in Sections 5.3 and 5.4. Section 5.3 presents dynamic programming scheme for partitioning a sorted rectangle set. In Section 5.4 we introduce a query adaptive space filling curve based on Z-Curve. We report results of extensive experiments in Section 5.5.

5.1 Preliminaries

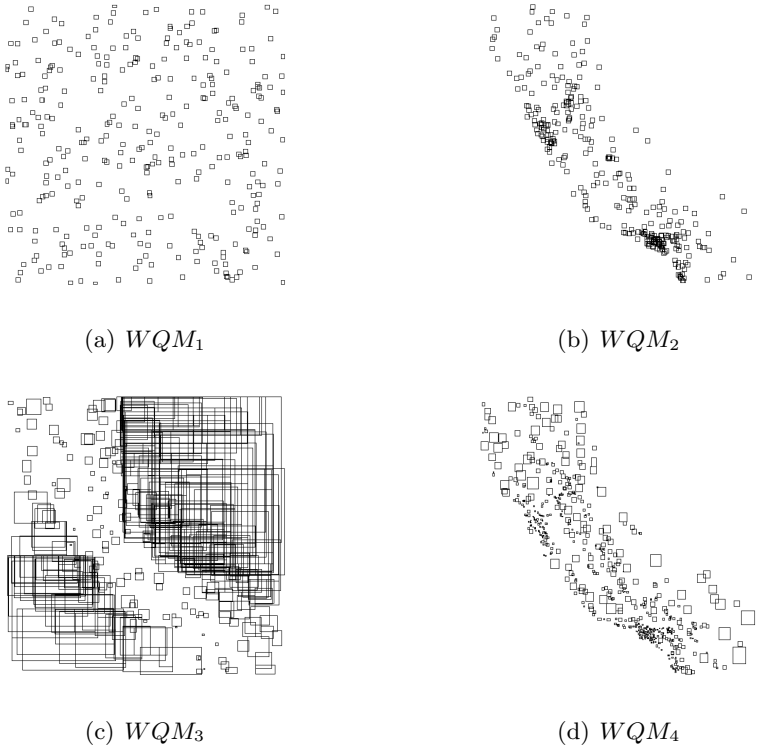
Query profile QP provides a (statistical) model that is derived from a collection of representative queries Q. For brevity, we assume input data set $R = \{r_1, \ldots, r_N\}$ as well as set Q of range queries are represented by d-dimensional rectangles. For example QP contains representative shape of a query defined by average side lengths and aspect ratios. We also consider the case where QP is unknown (arbitrary range queries). Our goal is to obtain an R-tree that minimizes average I/O costs for a class of ranges queries defined by QP. Our default query profile derived from set Q is defined as follows:

Definition 1. *The default query profile $QP = s_1, \ldots, s_d$ derived from the non-empty set of range queries Q is defined as follows: $QP[i] = s_i$ where s_i is an average side length for dimension i obtained from the set of range queries Q.*

For $d = 2$, $QP = [sx, sy]$, where sx and sy is the average size of the range query in the first and second dimension, respectively. For the sake of simplicity, we consider a two-dimensional space ($d = 2$) throughout this chapter. We will discuss the generalized case for $d > 2$ when necessary.

In order predict the average query performance of R-tree without constructing the index, the following cost models were proposed in a literature [73, 115, 95]. We briefly introduce the most common one [95]: The authors classified range queries according to the indicators *aspect ratio*, *location* and *size*. The query size is defined by either area (relative to the entire data space) or the number of qualified objects. Query location can follow either a uniform distribution or the distribution of the underlying data. The aspect ratio equals the width-to-height ratio of the query rectangle, which we assume to be 1 (quadratic windows) in the following. This yields in four different query models:

- WQM_1: size = area, location = uniform distribution,
- WQM_2: size = area, location = data distribution,



(a) WQM_1 (b) WQM_2

(c) WQM_3 (d) WQM_4

Figure 5.3: Figures a-d illustrate query models $WQM_1 - WQM_4$ for California strees data set

- WQM_3: size = number of answers, location = uniform distribution,
- WQM_4: size = number of answers, location = data distribution.

Figure 5.3 displays four query models for a rectangle set California streets (see Figure 5.1). Hereafter we assume WQM_1 model as default: We assume that the domain corresponds to the two-dimensional unit square $[0,1)^2$. A rectangle $r_i = (cx_i, cy_i, dx_i, dy_i)$ is represented by its center (cx_i, cy_i) and its extension (dx_i, dy_i). For a window query $WQ_{q,s}$ given by its center $q = (qx, qy)$ and its extension $s = (sx, sy)$, the probability of a rectangle r_i intersecting the window is $(dx_i + sx) \cdot (dy_i + sy)$ (see Figure 5.4 where a rectangles r_i is extended with query side length). More precisely: the probability of intersecting the rectangle r_i is a fraction of extended area to the area of the space, since the area of the unit cube is equal to one we obtain the result. The average number of

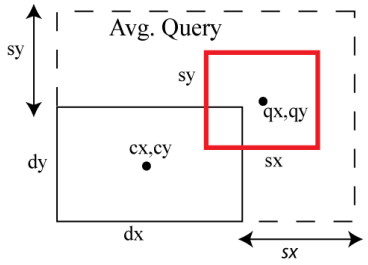


Figure 5.4: Figure illustrates C_{QP} model

rectangles intersecting the query window is then given by:

$$C_{QP} = \sum_{i=1}^{N} (dx_i + sx) \cdot (dy_i + sy) \tag{5.1}$$

Note that for point queries with $s = (0, 0)$, the equation computes the sum of MBR volumes. We obtain the expected number of leaf accesses, which is a typical performance indicator for R-trees, by applying the equation to the set of bounding boxes of the leaves. The essence of the cost model presented above is that minimizing the sum of area of the leaf nodes MBR induces less query I/O costs. We define the problem of query adaptive loading of R-trees as follows:

Definition 2. *Query adaptive loading: for a given query profile QP construct an R-tree such that C_{QP} is minimized.*

The leaf level nodes of an R-tree correspond to a partitioning of an input set. The partitioning consists of non overlapping buckets with capacity constraint. The minimal allowed number of entries per bucket corresponds to a minimal page capacity b and the maximal allowed number of entries to a page capacity B, respectively. According to a cost model a partitioning with the minimal cost yields better I/O performance. Thus, the problem of query adaptive loading is the problem of computing such partitioning.

Let us consider the case $s = (0, 0)$ (e.g. the query set is empty or unknown). Thus, we need to compute a partitioning such that the sum of MBR areas is minimized. Peter Widmayer showed that computing such partitioning is NP-hard [7]. We briefly review the main result. Let $P_{b,B} := p_1, \ldots, p_m$ be a partitioning consisting of m buckets. The capacity of bucket p is constrained by $b \leq |p_i| \leq B$ rectangles. Let $MBR(p)$ be a minimal bounding rectangle (box) of bucket p. We define $w : p_i \to \mathbb{R}^+$ a weight of bucket p. According to the cost model, the weight function $w := V(MBR(p))$ is an $MBR(p)$'s volume (for $d = 2$ its area). Thus,

Theorem 5.1. *The problem of partitioning $P_{b,B}$ for N given rectangles that minimizes $\sum_{i=1}^{n} V(MBR(p_i))$ is NP-hard.*

The proof considers the special case of $B = 3, b = 2$ (as $b = \lceil B/2 \rceil$) and a 2-dimensional space. Widmayer uses a polynomial time reduction from the version of planar 3SAT [81, 120] problem in which for each variable, also an edge can be embedded in the plain, to show NP-hardness of the problem. We refer for details of the proof to [7]. Based on these results, we develop a heuristic approach that optimally solves the partitioning problem for a given sorting order and $V(MBB(p))$. The justification for a heuristic approach lies in the NP-hardness of the problem.

As the target optimization function is obtained from WQM_1 query model, we assume that queries, more precisely their centers, are uniformly distributed in the underlying domain. This assumption is obviously not satisfied in a real application. The standard approach to overcome this deficiency is to use multidimensional histograms and to maintain these parameters for each histogram cell independently [10, 99]. This approach has already been used successfully for the analysis of R-trees [115].

5.2 R-tree Bulk-Loading Framework

For a given query profile $QP = (sx, sy)$, our goal is to generate R-trees whose average number of leaf accesses is minimized for queries derived from QP, as they dominate the overall cost for sufficiently large range queries. Moreover, upper levels of the trees are often located in memory, while leaf pages are generally not.

Our goal is to create optimal R-trees level by level, bottom-up. However, as shown in [7], the problem of generating optimal R-trees is NP-hard and, therefore, sort-based heuristics are examined traversing the following five steps:

1. **Determination of Sort Order:** For a given QP determine a sort order that minimizes the cost.
2. **Sorting:** Sort the rectangles with respect to the determined order.
3. **Partitioning:** Partition the sorted sequence into subsequences of size between b and B and store each of them in a page.
4. **Generation of Index Entries:** For each page, compute the bounding box of its partitions and create the corresponding index entry.

5. **Recursion:** If the total number of index entries is less than B, store them in a newly allocated root. Otherwise, start the algorithm with the generated index entries (bounding boxes) from Step 4.

Step 2 and Step 4 are very similar to the traditional sort-based loading of R-trees [104]. The crucial optimization occurs in the first and third step. Step 1 computes a sort order from the query profile. We exploit the fact that a space-filling curve (SFC) does not require a symmetric treatment of dimension, but allows more flexibility [93]. As an example, consider partial exact match queries orthogonal to the x-axis. Then the sort order should be only influenced by the x-value. This corresponds to a SFC where all bits of the x-axis should precede the bits of the y-axis. In step 4, the rectangles are then assigned to pages such that the capacity constraints of the R-tree are met. Filling up pages to the maximum (or as generally suggested to a constant degree) does not lead to R-trees optimized in respect to the given query profile. High storage utilization is only useful for fairly large queries, while the performance of smaller queries suffer. In Section 5.3, we present a heuristic partitioning algorithm that is optimized according to the underlying query profile. Step 3 as well as Step 4 make use of a cost model that is derived from our query profile.

5.3 Sorted Set Partitioning

In this section we consider the problem of query-adaptive partitioning a sorted sequence $r_1, \ldots, r_N$ of rectangles such that each bucket of the partition corresponds to a page of the R-tree. This approach is a heuristic that is based on the specific sorting order, since the computation of an optimal partition is NP-hard for $V(MBR(p))$. Every bucket corresponds to a contiguous subsequence $p_{i,j} = r_i, \ldots, r_j$ such that $b \leq j - i + 1 \leq B$ is satisfied. A valid partition P consists of the subsequences $p_{i,j}$ such that each rectangle belongs to exactly one of them. Let S_N denote the set of all valid partitions and let $S_{N,m}$ be the partitions that consist of exactly m buckets. While the standard sort-based loading strategy stores a fixed number of rectangles per page, we do not require equal numbers of objects per pages in our approach. This gives us flexibility to optimize the partition according to a given query profile QP. Let $MBR(p_{i,j})$ be the minimal bounding rectangle of a contiguous sequence $p_{i,j}$ of rectangles.

Based on the cost model (see Equation 5.1) we consider the following optimization problems:

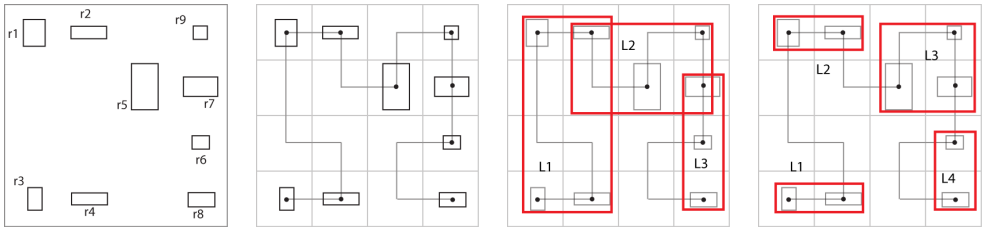
1. **Storage-bounded partitioning:** Compute a partition $Sm_{opt} \in S_{N,m}$ that minimizes the cost function for the set $\{MBR(p) | p \in S, S \in S_{N,m}\}$.

notations	description
S	a partitioning obtained from sorted sequence $r_1 \dots r_N$ such that each rectangle belongs to exactly one bucket $p_{i,j} \in S$. Each bucket $p_{i_j} \in S$ corresponds to a contiguous subsequence $p_{i,j} = r_i, \dots, r_j$ such that $b \leq j - i + 1 \leq B$ is satisfied.
$MBR(p_{i,j})$	minimal bounding box (rectangle) of a bucket $p_{i,j} \in S_{b,B}$
S_N	set of all valid partitions obtained from a sorted sequence $r_1 \dots r_N$
$S_{N,m}$	a set of valid partitions with exact m buckets obtained from a sorted sequence $r_1 \dots r_N$
$f_w : p_{i,j} \to \mathbb{R}^+$	weight function for buckets $p_{i,j} \in S_{b,B}$
$V^+(MBR(p_{i,j}), QP)$	MBR volume extended with the average side length from QP, for $k = 2$: $QP = (sx, sy)$ and $V^+(MBR(p_{i,j}), QP) = (dx + sx) \cdot (dy + sy)$
$C(S) = \sum_{p_{i,j} \in S} f_w(p_{i,j})$	cost function of partitioning S
$C_{QP}(S) = \sum_{p_{i,j} \in S} V^+(MBR(p_{i,j}), QP)$	default cost function $C(S)$

Table 5.1: Important notations

2. **Query-optimal partitioning:** Compute a partition $S_{opt} \in S_N$ that minimizes the cost function for the set $\{MBR(p) | p \in S, S \in S_N\}$.

Figure 5.5 depicts two possible partitioning with different $C(S) = \sum_{p_{i,j} \in S} V(MBR(p_{i,j}))$ costs (sum of MBRs areas) with parameters $b = 2, B = 3$. Note, that this cost function corresponds to the case where the index should be optimized for a point queries. The center point of rectangle is used for mapping to the Hilbert key. Input rectangles $(r_1, \dots, r_9)$ (see Figure 5.5(a)) are processed according to Hilbert curve (see Figure 5.5(b)). Figures 5.5(c) and 5.5(d) show partitioning obtained by applying standard fixed sized partitioning and Query-optimal partitioning, respectively. The partitioning in Figure 5.5(c)



(a) (b) (c) (d)

Figure 5.5: Sorted set partitioning

consists of three buckets (r_3, r_4, r_1), (r_2, r_5, r_9) and (r_7, r_6, r_8). The partitioning with four buckets (r_3, r_4), (r_1, r_2), (r_5, r_9, r_7) and (r_6, r_8) in Figure 5.5(d) display lower C costs (the sum of MBRs areas). Therefore, the partitioning with the same number of rectangles per bucket not always lead to a minimal cost partitioning. Though, the partitioning in Figure 5.5(d) has more buckets, it displays better costs. Moreover, the worst case number of produced buckets is still limited by N/b.

Note that query-optimal partitioning results in a better partitioning, but the worst-case, storage utilization of the resulting R-trees can be as low as b/B. Storage-optimal loading allows us to choose the desired storage utilization $(N/(m \cdot B))$ in advance by setting m.

Let $QP = (sx, sy)$ be a given query profile and $C_{QP}(S) = \sum_{p \in S} V^+(MBB(p), QP)$ be the sum of areas extended with average side length from query profile QP. More formally, $V^+(r, QP) = (dx + sx) \cdot (dy + sy)$ for a rectangle $r = (cx, cy, dx, dy)$. $C_{QP}(S)$ denotes denotes the cost of a partition $S \in S_N$ for a given query profile QP. This function has a nice property, it allows us designing of an efficient algorithm to compute the optimum. Consider a split of a partition S into two arbitrary partitions S_l and S_r. Then, the following property holds for our cost function:

$$C(S) = C(S_l) + C(S_r)$$

In particular, equality is satisfied for the optimal partition S_{opt}. Note that, S_l and S_r must also be optimal partitions of their associated rectangles. In fact, this observation allows us to use the paradigm of dynamic programming in a similar way as for computing optimal histograms [72, 120].

For partitioning the first i rectangles into k contiguous sequences, the computation of

the minimum cost $opt^*(i,k)$ can be expressed by the following recursion:

$$opt^*(i,k) = \min_{b \le j \le B} \{opt^*(i-j, k-1) + f_w(p_{i-j+1,i})\} \tag{5.2}$$

Algorithm 5: $opt^*(i,k)$

Input: $R[\,]$ array of rectangles of size N, f_w weight function, m, b, B
Output: $cost[1 \ldots N][1 \ldots m]$ cost array

```
1  allocate cost array, and initialize for one node;
2  cost[ ][ ] ;
3  for i = b to B do
4      cost[i][1] = f_w(MBR(p_{1,i}));
5  compute best costs for m nodes starting from 2 ;
6  for y = 2 to m do
7      assignment to y pages ;
8      for x = y · b to min(y · B, N) do
9          s[b ... B] = 0;
10         max number of entries per node;
11         max_B = (x − B > 0)?B : x − B + 1 ;
12         R_p[ ] ← precompute MBRs costs for t-B to t-b ;
13         r ← rectangle ;
14         for i = 1 to max_B do
15             r ∪ R[x − i] compute MBR ;
16             if i ≥ b then
17                 R_p[i] ← f_w(r);
18         for l = b to max_B do
19             s[l] = cost[x − l][y − 1] + R_p[l − b + 1];
20         cost[x][y] = min(s);
21 return cost[ ][ ];
```

In general, the $opt^*(i,k)$ function corresponds to the optimal one-dimensional histograms computation proposed by [72] if we use set $b = 1$ and $B = N - m - 1$. In order to compute $opt^*(N,m)$, we apply the recursive formula for all $1 \le i \le N$ and $1 \le k \le m$, in increasing order of k, and for any fixed k, in increasing order of i. We store all computed values of the $opt^*(i,k)$ in a table (see Alg. 5). Thus, when a new $opt^*(i',k')$ is calculated using Equation 5.2, any $opt^*(i,k)$ that may be needed can be read from the table. After computation of the optimal cost, we can read the contiguous sequences of the input rectangles out from the dynamic programming table. From this

procedure, we obtain the following result.

Theorem 5.2. *An optimal partition $S_{N,m}$ of N rectangles into m buckets, each of them containing between b and B contiguous rectangles, can be computed in $O(N \cdot m \cdot B)$ time and $O(N \cdot m)$ space for weight function $f_w := V^+(MBR(p_{i,j}), QP)$.*

Proof. For each k-buckets we have $k \cdot (B - b)$ subproblems. Thus, the overall number is $\sum_{k=1}^{m} k \cdot (B - b) \leq B \sum_{k=1}^{m} k \leq O(B \cdot m^2) = O(N \cdot m)$, since in our settings $m \in O(N/B)$. Therefore, we need at least $m \in O(N/B)$ space in order to obtain the optimal partitioning via backtracking. Once we have computed optimal solution for $k - 1$ buckets, we have $B - b = O(B)$ choices to extended our optimal solution with k-th bucket. Thus, at least $O(N \cdot m \cdot B)$ time we need to fill up the dynamic programming table. Since our weight function is an extended area, it can be computed in $O(1)$ time for a given rectangle (MBR). Yet, we need to compute f_w for a $B - b = O(B)$ MBRs for each subproblem. Since B MBRs for B choices are computed in $O(B)$ time (by executing MBR union operation iteratively), we obtain our overall time costs. □

Algorithm 5 provides details of $opt^*(i, k)$ computation. For each k we consider $k \cdot (B - b)$ positions in the input array. Therefore, we compute B MBRs for each position i several times, since position intervals can overlap for different k values. Although we do not improve asymptotically $O(N \cdot m \cdot B)$ time, Authmann [32] proposed runtime improvement of the algorithm using additional $N \cdot (B - b)$ space by precomputing for each i the costs of MBRs of $B - b$ buckets $p_{i,i-b} \ldots p_{i,i-B}$.

Next, we consider query-optimal loading, the problem of computing the optimal partition without user-defined storage utilization. At first glance, the problem appears to be harder because the solution space is larger. However, the opposite is true because the parameter m has no effect on the optimal solution anymore. This results in the following simplified recursion:

$$gopt^*(i) = \min_{b \leq j \leq B} \{gopt^*(i - j) + f_w(p_{i-j+1,i})\} \tag{5.3}$$

In order to compute $gopt^*(N)$, we compute the recursive formula for all $1 \leq i \leq N$ in increasing order of i. We store all computed values of $gopt^*(i)$ in a table (see Algorithm 6). Thus, when a new $gopt^*(\acute{i})$ is calculated using Equation 5.3, any $opt^*(i)$ that may be needed can be read from the table. As in the case for opt^*, we obtain the result sequences from the table. Thus, the following theorem holds:

Theorem 5.3. *An optimal partition S_N of N rectangles into buckets, each of them containing between b and B contiguous rectangles, can be computed in $O(N \cdot B)$ time*

Algorithm 6: $gopt^*(i)$

Input: $R[\,]$ array of rectangles of size N, f_w weight function, b, B
Output: $cost[1 \dots N]$ cost array

```
1  cost[ ]  allocate cost array, precompute costs for 1 to B elements ;
2  for  t = 2b to N do
3      cost[t] ← ∞ ;
4      R_p[ ] ←  precompute MBRs costs for t-B to t-b ;
5      r ← rectangle ;
6      for i = 1 to B do
7          r ∪ R[x − i] compute MBR ;
8          if i ≥ b then
9              R_p[i] ← f_w(r);
10     for  l = B to b do
11         compute cost for last b to B elements if t-l > b;
12         c_p ← get MBR costs R_p[l − b + 1];
13         c_p ← cost[t − l] + c_p ;
14         if c_p < cost[t] then
15             cost[t] = c_p;
16 return cost[ ];
```

and O(N) space for weight function $f_w := V^+(MBR(p_{i,j}), QP)$.

Proof. An optimal solution for position i uses only one supbroblem that is at most B and at least b positions away. Thus, the overall number of subroblems is $O(N)$. Therefore, we need at least $O(N)$ space in order to obtain an optimal partitioning via backtracking.

At each position i we have to make $B - b$ choices to extend an optimal solution. Since our weight function is an extended area, it can be computed in $O(1)$ time for a given rectangle (MBR). Yet, we need to compute f_w for a $B - b = O(B)$ MBRs for each subproblem. Since B MBRs for B choices are computed in $O(B)$ time (by executing MBR union operation iteratively), we obtain our overall time costs. □

Theorem 5.3 shows that optimal loading is possible in as little as linear time. The required CPU-time is much lower compared to the optimal solution of space-bounded loading. Note that storage utilization of R-trees generated by query-optimal loading largely depends on the underlying query profile. If the query size is large, the optimal partitioning also causes high storage utilization.

Note, that opt^* and $gopt^*$ compute only the best partition for a given sequence for one level at time. To build an optimal R-tree that include all levels, we can generalize $gopt^*$

for $k-$levels. As for *gopt* we compute the best partitioning for subsequences of size $\Theta(B^l)$ for levels $l = 1, \ldots$. To limit the processing time and bound number of entries per node in upper tree levels, we adapt the approach known from weight balanced B-trees [28]. We define parameter a as a branching parameter. Let $b = 1/3B, a = 1/4B$ and $l = 1 \ldots$ then following function computes partitioning:

$$g^*(i,j) = \begin{cases} f_w(p_{i,j}) \quad \text{if } \frac{B}{3} \leq (j-i) \leq B \text{ and } 0 \leq i < j \leq N \\ \min\limits_{\frac{1}{3}B\frac{(j-i)}{a} \leq k \leq \frac{4}{3}B\frac{(j-i)}{a}} \{g^*(i,j-k) + g^*(j-k+1,j) + f_w(p_{j-k+1,j})\} \quad \text{if } 0 \leq i < j \leq N \\ \infty \quad \text{otherwise} \end{cases} \tag{5.4}$$

As for *gopt** and *opt** we use a table to hold intermediate costs. Thus, the following theorem holds[2]

Theorem 5.4. *An optimal weight-balanced R-tree with capacity parameters B, b and branching parameter $a = \frac{1}{4}B$ can be computed from a sorted sequence of rectangles in $O(N^3 \cdot \frac{1}{4}B^2 \cdot (\frac{1}{4})^{\log_B(N)})$ time and $O(N^2)$ space for weight function $f_w := V^+(MBR(p_{i,j}), QP)$.*

Proof. The space needed for a dynamic programming table is $O(N^2)$, since we need to get level partitioning via backtracking. As the number of sub-sequences is bounded by $O(N^2)$ for N elements, all (i,j) interval costs belonging to sub roots of particular level are obtained from this table.

The computation time can be derived as follows: first we derive the complexity of minimal cost computation from a table (inner loop see Algorithm 7) depending on the level. For the level 0 the inner loop has the complexity $B - \frac{1}{3}B \leq B$. For the level $l > 0$ the inner loop has the following complexity $\frac{4}{3}B(\frac{1}{4}B)^l - \frac{1}{3}B(\frac{1}{4}B)^l = B(\frac{1}{4}B)^l$. Summing up over all levels we get $O(B \cdot \sum_{k=0}^{l} (\frac{1}{4}B)^l)$. Since i and j define valid partition bounds of size $O(B^l)$ for each level l we invest at most $O(N^2)$ operations per level. Then we have following overall costs since $l \leq \log_B N$ $O(N^2 \cdot B \cdot (\frac{1}{4}B)^{l+1}) \leq O(N^3 \cdot \frac{1}{4}B^2 \cdot (\frac{1}{4})^{\log_B(N)})$. □

If we use a non-weight balanced variant then the inner loop for a level l has at least complexity $B^{l+1} - (\frac{1}{3}B)^{l+1}$. The computation time increases almost by a factor four compared with a weight-balanced variant per level, since: $B^{l+1} - (\frac{1}{3}B)^{l+1}/(B(\frac{1}{4}B)^l) = (1 - (\frac{1}{3})^l)/(\frac{1}{4})^l$ Thus, for $l \leq \log_B N$ weight-balancing decreases the overall computation time by almost factor $4^{\log_B(N)}$.

[2] In [7] we devised $O(\log_B N \cdot N^2)$ space bound. As we can decode each partition interval i, j for each level using N^2 space, we provide more tight space bound in this work.

Algorithm 7: $g^*(i,j)$

Input: N rectangles, f_w weight function, b, B, a
Output: $cost[1 \dots N]$ cost array

```
1  cost[ ][ ]  allocate cost array;
2  precompute F_w[ ][ ] array ;
3  for i = 1 to N − b do
4      r ← rectangle ;
5      for j = i to N do
6          r ∪ R[j] compute MBR ;
7          F_w[i][j] ← f_w(r) ;
8  F_w[ ][ ] contains the costs of MBRs for all i, j ;
9  for each level compute best partitioning ;
10 for  l = 0 to ← ⌈log_a N⌉ do
11     s ← b · a^l ;
12     e ← B · a^l ;
13     for i = 0 to N − b · a^l do
14         for j = i + b · a^k to N do
15             for  k = s to e do
16                 if l == 0 then
17                     c[i][j] ← F_w[i][j] ;
18                 else
19                     c_p ← F_w[j − k + 1][j] ;
20                     c_p ← c[i][j − k] + c[j − k + 1][j] + c_p ;
21                     if c_p < c[i][j] then
22                         c[i][j] = c_p ;
23 return cost[ ];
```

As a result, a subtree on level $l > 0$ holds $\left[\frac{1}{3}B(\frac{B}{4})^l, \frac{4}{3}B(\frac{B}{4})^l\right]$ elements in its leaf nodes and $\left[\frac{B}{16}, B\right]$ entries per index node. However, we need at least a quadratic space and time for a computation, since all possible subsequences should be considered. Consequently, the solution is not practical anymore. In our experiments we observed only a marginal improvement in comparison to *gopt**, since only a small subset of a data can be processed efficiently at time.

5.3.1 Practical Considerations

In the following, we provide some useful information for processing a large set of rectangles. Because computing *opt** requires quadratic space, it is unlikely that the whole intermediate data sets can be processed in memory. In this case, the data set is processed as follows: we cut the data in sufficient big equi-sized chunks and apply *opt** on each of them independently. In our experiments, we observed that B^2 (where B is equal the number of rectangles in a page) is sufficient to obtain near-optimal results. For the computation of *gopt**, the same strategy can be applied. However, since only the last B entries are required by *gopt**, a buffer of B entries is sufficient for processing.

After the first level has been constructed, the index entries of the next level can be resorted again. However, we noticed that for a given query profile, the produced sequence of MBRs already preserves the order of the input rectangles so that we skip the extra sorting step to reduce the total build-up time.

5.4 Optimization of Sort Order

The quality of our partitioning algorithms depends on the chosen sorting order. Our experiments show that traditional Hilbert and Z-Curve perform very well in combination with the proposed partitioning for square query rectangles. In this section we provide an algorithm for determining the sorting order of our bulk-loading framework for the cases where the average query shape is non-square. The sorting order is defined by a SFC whose input corresponds to an appropriate shuffling of d bit sequences, where each of them of constant length L represents a dimension of the d-dimensional unit cube $[0, 1)^d$. As before, we assume two-dimensional data and discuss the general case only if necessary. Due to its flexibility, we use the Z-curve as our SFC in the following.

Our goal is to adapt to the underlying query profile $QP = (sx, sy)$. In order to model non-square window queries, we introduce here the *aspect ratio* given by $a = sy/sx$. The effect of the aspect ratio is illustrated in Fig. 5.2(a) where a set of range queries with a high aspect ratio is plotted. The bounding boxes of the R*-tree leaves are plotted in Fig. 5.2(b), while the plot of the boxes obtained from our sort-order optimized algorithm is given in Fig. 5.2(c). The R*-tree does not take any query profile into account and attempts to generate boxes with a quadratic shape, while our new loading algorithm adapts its boxes to the shape of the query. This query-adaptive partition causes an substantial improvement in performance compared to the standard R*-tree.

The basic idea is to introduce a two-part SFC. The first part corresponds to a SFC being defined on a non-symmetric binary grid. Each dimension of a grid is partitioned

in binary manner into equi-sized intervals. The grid resolution GR is given by the total number of bits required for determining whether a point belongs to a cell. Note that the volume of the cell is 2^{-GR}. The second part combines the remaining $d \cdot L - GR$ bits in lexicographic order. Consider an example shown in Figure 5.6. We assume that we use four bits to represent a singe dimension, and we have 256 rectangles. We can represent 256 Z-addresses corresponding to a one cell. We assume that each object is mapped to exactly one distinct address. Now let us assume that a capacity of a page is four objects. For quadratic queries with volume of four cells, the sorting order that corresponds to symmetric Z-Curve $\underbrace{xyxyxyxy}_{GR=8}$, minimizes number of the node access. Symmetric Z-curve enforces generation of square shaped MBRs. In the case of non-square query shapes with same volume, the asymmetric Z-curve $\underbrace{xyxy}_{GR=4}xxyy$ enforces generation of MBRs with the same aspect ratio as the query rectangle. Note that this design of the two-part SFC allows us adapting to the common cases discussed previously. In the case of $a = 1$, we fully exploit the first part of our SFC, i.e., $GR = d \cdot L$, while for partial match queries, we only use the second part with an appropriate lexicographic order (given priority to the most selective dimensions). The fundamental questions are how the asymmetry of

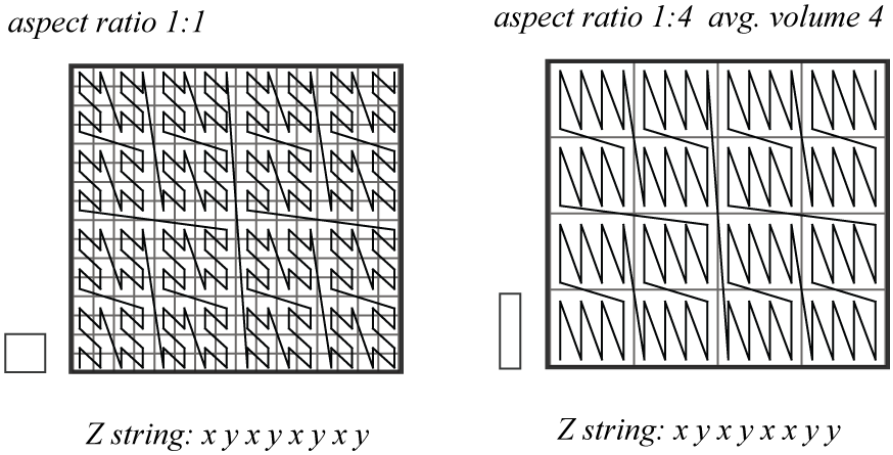


Figure 5.6: Left a grid with $GR = 8$, right a grid with $GR = 4$.

the grid is determined and how GR has to be set for a given query profile. Our goal is to design a grid such that the number of grid cells is minimized given that the volume $V = x \cdot y = 2^{-GR}$ is fixed. Here, x and y denotes the size of bounding intervals of the cell. Let a query profile be $Q = (sx, sy)$, with $sy = a \cdot sx$. These simplified assumptions allows us to use Equation 5.1 for estimating the average number of cells intersecting a

window query. The $LC2(x, y)$ expresses the number of cells as a function of x and y.

$$LC2(x, y) = 2^{GR} \cdot (x \cdot y + x \cdot sy + y \cdot sx + sx \cdot sy) \tag{5.5}$$

Equation 5.5 can be rewritten by substituting x by V/y, sx by $\frac{sy}{a}$ and $x \cdot y$ by the constant V. Note that the average utilization is constant for different sort orders. This provides the following cost function:

$$LC(y) = 2^{GR} \cdot (V + sy \cdot (\frac{V}{y} + \frac{y}{a}) + sy^2 \cdot a) \tag{5.6}$$

$$LC^{'}(y) = 2^{GR} \cdot sy \cdot (\frac{1}{a} - \frac{V}{y^2}) \tag{5.7}$$

Computing the root of the derivative of equation 5.6 yields the minimum. It directly follows that $y_{opt} = \sqrt{V \cdot a}$ minimizes $LC(y)$. In addition, we obtain $x_{opt} = \sqrt{\frac{V}{a}}$ and that the aspect ratio of the optimal cells is also equal to a again. Note that we ignore here that our optimum is not realized on the grid and some rounding is actually necessary.

In the case of $d > 2$, we introduce $d - 1$ aspect ratios $a_1, \ldots, a_{d-1}$ with $a_i = \frac{s_{i+1}}{s_i}$. Let $V = \prod_{1 \leq i \leq d} x_i$ be the average volume of a page region and x_i be the length of the i-th side of the page region. Then, LC is minimized for $x_d = (V \cdot a_{d-1})^{1/d}$, $x_i = (V \cdot \frac{a_{i-1}}{a_i})^{1/d}$ for $1 < i < d$ and $x_1 = (\frac{V}{a_1})^{1/d}$.

Let us now discuss how to set the parameter GR or equivalently the concrete size of a grid cell. There are at least two intuitive options. One is to set the average query volume equal to the average query size. Then a query hits at most four cells. As shown in [31], this minimizes the number of contiguous pieces of the SFC that intersect the query region. However, our goal is to minimize node accesses, thus we use the average size of the optimal bounding boxes of R-tree leaves (which means the optimal one obtained from the cost function 5.1) to determine the grid cell (see Fig 5.6). The results of our experiments indicate that this option is superior to the first option. Note that the optimal bounding box offers the same aspect ratio as the window query. We use this property to initialize our algorithm with this box rather than using $d - 1$ aspect ratios and the parameter GR (see Alg. 8 for details). The input of AdaptiveShuffle consists of a d-dimensional array A of bit sequences of fixed length L and a d-dimensional array len representing the shape of the optimized boxes. A_i denotes the value of the i-th dimension. In order to simplify the description of the algorithm, we assume that $len_i < len_{i+1}$ is satisfied, $1 \leq i < d$, without loss of generality. Each part of the two-step SFC consists of a for-loop. In the first for-loop, the routine $SymShuffle$ shuffles a certain number

Algorithm 8: Algorithm AdaptiveShuffle

Input: Average edge lengths of the boxes of the leaves $(len_1, \ldots, len_d)$ with $len_i \leq len_{i+1}$, d-dimensional array A of bitstrings with L bits per bitstring
Output: bitstring of length $L \cdot d$

```
1 from = L, resString = ∅;
2 for k = d, ..., 1 do
3 |   to = L − ⌈log_2 1/len_k⌉;
4 |   resString = +SymShuffle(A, k, from − 1, to);
5 |   from = to;
6 for k = 1, ..., d do
7 |   resString = +SuffixString(A_k, L − ⌈log_2 1/len_k⌉);
8 return resString ;
```

of bits of the first k dimensions in a symmetric manner until the selectivity of the k-th dimension is fully exploited. The symbol "=+" denotes appending the right string to the result string. This loop is iteratively performed for $k = d, d-1, \ldots, 1$. Note that the parameter GR can be computed by $GR = \sum_{1 \leq k \leq d} \left\lceil \log_2 \frac{1}{len_k} \right\rceil$. The second for-loop simply calls $SuffixString$ to append the unused bits of the k-th dimension to the result string, $k = 1, \ldots, d$. Let us consider an example for $d = 3$, $L = 6$, $A_1 = (x_5, \ldots, x_0)$, $A_2 = (y_5, \ldots y_0)$, $A_3 = (z_5, \ldots, z_0)$ and $len = (\frac{1}{16}, \frac{1}{8}, \frac{1}{2})$. From these settings, we obtain the following result string:

$$x_5, y_5, z_5, x_4, y_4, x_3, y_3, x_2, x_1, x_0, y_2, y_1, y_0, z_4, z_3, z_2, z_1, z_0$$

Note that we first interleave bits from all dimensions. After the first cycle, the z-dimension is not involved anymore. After three cycles, the asymmetric grid with resolution $GR = 8$ is generated and the remaining bits are then simply appended to the result.

5.5 Experiments

In this section, we compare different sort-based loading algorithms in a set of experiments and show the improvements of our query-adaptive technique. We first describe data files and query sets used in our experiments. Then, we present improvements achieved by our algorithms and compare the influence of order optimization and the partitioning strategies on both our and also related loading algorithms. In addition, we discuss the

validity of our assumptions, which have influenced the design of our loading algorithms.

5.5.1 Data File and Query Profiles

In our experiments, we adapted the test framework developed for RR*-tree evaluation [39]. The framework consists of 28 different data sets, either points or rectangles, that belong to eight groups *abs, bit, dia, par, ped, pha, uni, rea.* Each of the first seven groups contain three artificially generated data sets with 2,3, and 9-dimensional data following the same distribution in each dimension. Each of the artificial data sets contains at least 1 million objects from $[0,1]^d$. For example, the group *uni* consists of 3 files of $1'000'000$ two-, three- and nine-dimensional uniformly distributed points. We give a brief overview about the data sets; 2-dimensional data sets can be roughly grouped in two groups point sets and rectangular sets. Data set *abs* consists of equal sized squares generated from equidistant distribution. Data set *bit* is a point distribution generated according the power low and closely related to Zipf-distribution. Data set *dia* consists of rectangles distributed along the main diagonal. Data set *par* represents a rectangular distribution with a high variance of the size and the shape of rectangles. *ped* is a point distribution of a thin stripped clusters obtained from a data set *par*. Data set *pha* is a set of a ellipse shaped clusters of points generated from data set *par*. Data set *uni* is a uniform point distribution. The eighth group *rea* contains seven real data sets with 2,3,5,9,16,22, and 26 dimensions, respectively. For example, the 2-dimensional data set consists of 1'888'012 bounding boxes of streets of California. The 3-dimensional data set is contains 11'958'999 points from a biological application. The data sources as well as a full description of the data sets are available from [39].

In the original test framework, three range-query sets $qr1$, $qr2$ and $qr3$ were considered for each data set. Except for the group *ped*, the query sets were generated as follows: The queries of $qr1$, $qr2$ and $qr3$ refer to square-shaped windows and deliver 1, 100 and 1000 results on average, respectively. Note that in difference to previous performance comparisons, the cardinality of the response sets is limited (at most twice the average) to avoid the dominating influence of a few queries with very large response sets. All queries followed the underlying data distributions. According to the query taxonomy [95], these query sets are of type WQM_4 (queries follow data distribution and query size is based on answer number). For group *ped*, queries were generated in a more traditional way. The square-shaped range of $qr1$, $qr2$ and $qr3$ cover $k/1'000'000$ of the entire data space, k = 1, 100, 1000. In addition, *ped* queries were uniformly distributed (type WQM_1).

In order to examine the query-adaptivity of our techniques, we modified the generation of 2-dimensional query profiles $qr2$ and $qr3$ by introducing the aspect ratio a as a new

parameter. There are now $qr2_a$ and $qr3_a$, $a = 1, \ldots, 20$, where $a = 1$ refers to the original profiles. We retain the original methodology for generating query profiles $qr2_a$ and $qr3_a$ limiting the response set cardinality to 100 and 1000, respectively.

Except for *ped*, the generation process is based on posing nearest neighbor queries with the weighted distance measure $L_\infty(p1, p2) = \max(|p1x - p2x|, \frac{1}{a}|p1y - p2y|)$, $p1, p2 \in [0, 1]^2$. For *ped*, we considered range queries with query profile $(\sqrt{(k/(a \cdot 1'000'000))}, \sqrt{(a \cdot k/1'000'000)})$.

5.5.2 Examined Algorithms

Table 6.2 provides a summary all methods used. As a reference method, we used the traditional sort-based loading termed Z-loading and H-loading using Z-ordering and H-ordering, respectively. Both of the loading techniques are parameterized with storage utilization set to 80%. Note that in our experiments, higher storage utilization did not improve the query performance. ZAS-loading refers to Z-ordering combined with our adaptive shuffling technique. Z-GO stands for globally optimized partitioning technique applied to Z-ordered input, whereas H-GO is based on H-ordering. H-SO uses our partitioning with a guaranteed storage utilization of 80%.

We also examined STR [80] and TGS [58] because of their popularity. Storage utilization was again set to 80%. In addition, we also present an improved version of STR, termed STR-GO, which combines STR with our globally optimized partitioning method. STR-GO performs as STR for the first $d - 1$ dimensions, but uses our partitioning technique for the last dimension. This is directly applicable because the data objects are distributed among the leaf pages regarding the d-the dimension. The performance of bulk loaded R-trees and tuple-by-tuple loaded R*-trees[38] is also compared.

All algorithms are implemented in Java. Experiments were conducted on a 64 bit Intel Core2Duo (2 x 3.33 Ghz) machine with 8 Gb memory running Windows 7. In order to illustrate the performance on several different storage devices, we conducted experiments on a magnetic disk (Seagte ST35000418As), SSD (Intel X25) and in main memory. For experiments on disk and SSD, we used 4KB pages with a capacity B = 128 and minimum occupation b = 42 for d=2. For sorting, we used 10 MB of main memory. The raw I/O device interface is used to avoid the interference with other system buffers. For our in-memory experiments, we used different settings for the page capacity that was found to be the overall optimum: B= 12 and b = 4.

Algorithm efficiency is measured by I/O and CPU time. We consider the number of leafs touched during query traversal as a default I/O metric, however, we do not count repeated accesses to the same leaf. As confirmed in our experiments, this is a good

Shortcut	Sorting Order	Partitioning
Z	symmetric Z-order	naive
H	H-order	naive
ZAS	adaptive Z-order	naive
Z-GO	symmetric Z-Order	$gopt^*(i)$
ZAS-GO	adaptive Z-order	$gopt^*(i)$
H-GO	H-order	$gopt^*(i)$
H-SO	Hilbert-Order	$opt^*(i,k)$
STR	not applicable	naive
STR-GO	not applicable	$gopt^*(i)$
TGS	not applicable	n/a

Table 5.2: Algorithms

performance indicator, since index nodes are located in large main memories.

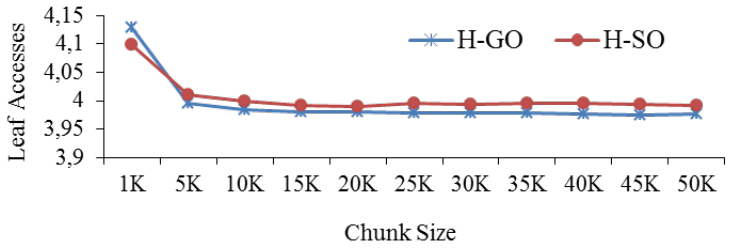


Figure 5.7: Query performance of partitioning algorithms for varying the chunk size

In Section 4.2 we introduced a simple approximation scheme for our partitioning algorithms. Rather than running the algorithm on the entire data set, we prepartition the data into equi-sized chunks and apply the algorithms to each of the chunks. Figure 5.7 depicts quality of the approximation as a function of chunk size for the California data set using $qr2$. We observed that a chunk size of B^2 $(= 16384)$ is sufficient to obtain near-optimal results. Similar results are achieved for other data sets. Note that the function is not decreasing strictly monotonically because the queries do not obey the uniform assumption of the query model. This also explains that for a chunk size of 1K the SO strategy is slightly superior to GO. For the rest of the experiments, we use chunks of size B^2 for our partitioning methods.

5.5.3 Sorted Set Partitioning

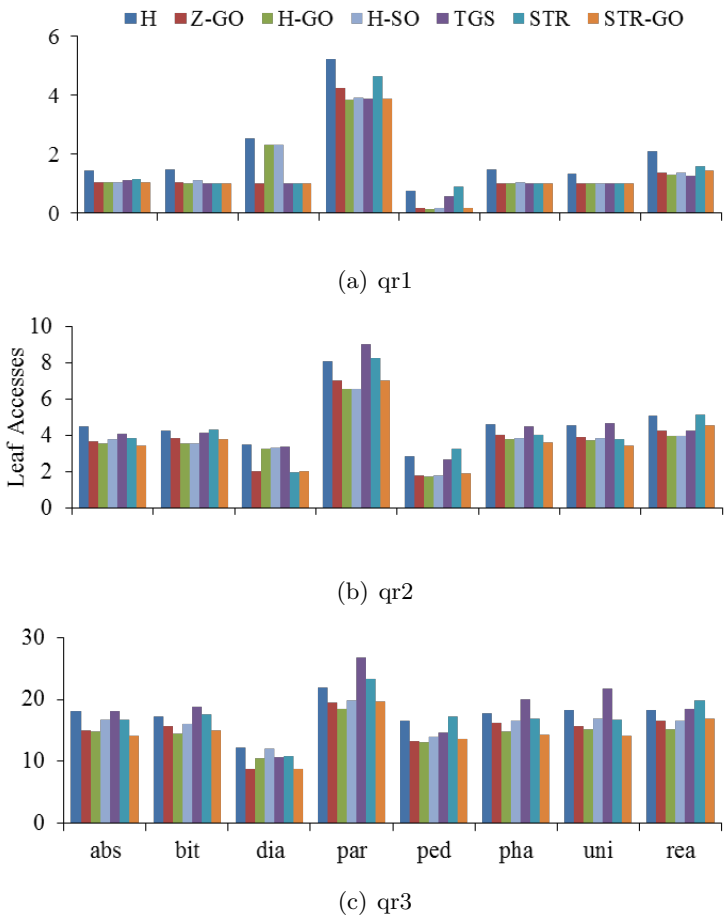
This section discusses the improvements achieved by our partitioning strategies. We consider square-shaped queries with aspect ratio $a = 1$ only. In addition to the methods

based on space-filling curve, we also report the results of TGS, STR and STR-GO. Figure 5.8 depicts the I/O performance for eight 2-dimensional data sets and query files $qr1$, $qr2$ and $qr3$. Note that all loading methods that use our partitioning strategies are superior to H-loading. Moreover, STR-GO performs better than its original counterpart. For TGS we observed similar effects as reported in [58]. TGS performs well for point queries $qr1$, but its performance deteriorates with an increasing query region. It is noteworthy that there is no significant difference between H-GO and Z-GO except for *dia*, where Z-GO is clearly superior. The most significant improvements over H-loading are achieved for point queries on the 2-dimensional data set *ped*. This data set is the only for which the queries are uniformly distributed. Note that this is in full agreement with the goal function used in our optimization. This also explains the large difference in performance between STR and STR-GO. We observed that the impact of the query size is marginal for storage bounded algorithms H-SO and Z-SO in comparison to the H-GO and Z-GO counterparts. Thus, minimizing the area (which is only optimal for point queries) achieves already good results for all query profiles $qr1$, $qr2$ and $qr3$.

The query size influences the relative R-tree performance. This is not surprising, as for larger regions, the storage utilization will have greater impact (than the clustering capability of the loading techniques). This is also in agreement with the analytical results obtained from the cost model. For example, R-trees generated from H-GO-loading perform small queries on the California data set (*rea*) with only 60% of the disk accesses compared to H-loaded R-tree. For queries $qr3$ with 1000 results the performance difference is only 20%. We achieved similar results for the 3-dimensional data sets. H-loading is superior to STR-GO for only some of the data files, but inferior to Z-GO and H-GO in all cases. The average normalized results for two, three and nine dimensions are reported in Table 5.3 (performance is expressed as the ratio of average number of leaf accesses for the specific and the H-loaded R-tree). The results indicate slight improvements for higher dimensions.

As expected, the number of leaf pages occupied by our R-trees generated from Z-GO and H-GO in relation to the number of leaves of H-loaded R-trees is higher for small queries. For larger queries, it is typically below 100%, i.e., the storage utilization is higher than 80% for the R-trees generated by Z-GO and H-GO.

Further, we analyzed average query execution time for d=2 and query file qr_2. Figure 5.9 shows the average time per query for disk, SSD and main memory. Query time measured for a disk includes the I/O time for leaf accesses, while the index nodes are likely to reside in memory or disk cache. In particular, we observed a positive effect of sort-based loading using H-loading combined with our partitioning on the average dis



(a) qr1

(b) qr2

(c) qr3

Figure 5.8: Avg. number of leaf accesses per query for d=2

access cost. The way how data is written to disk exhibits high clustering within a level, since blocks are written according to the SFC order. Therefore, there are fewer random I/Os than for TGS and STR (see Fig. 5.9). In order to illustrate the impact of physical clustering, we also compared the query performance with the R*-tree (see Fig. 5.10). As illustrated, significant improvements of up to factor of five can be achieved particularly because of the clustering when indexes are bulk-loaded. Moreover, we observed also similar effects for in memory R-trees. For SSDs, however, there are no positive effects from sequential I/O patterns. As a consequence, the average query time is highly correlated to the number of node accesses, see the plots in the mid of Figure 3 and 4.

In the following we discuss the effects of the uniformity assumptions of our cost model. Recall that except for *ped* the query distributions follow the data distribution. The question is therefore how our simplified analytical cost model is related to the real cost.

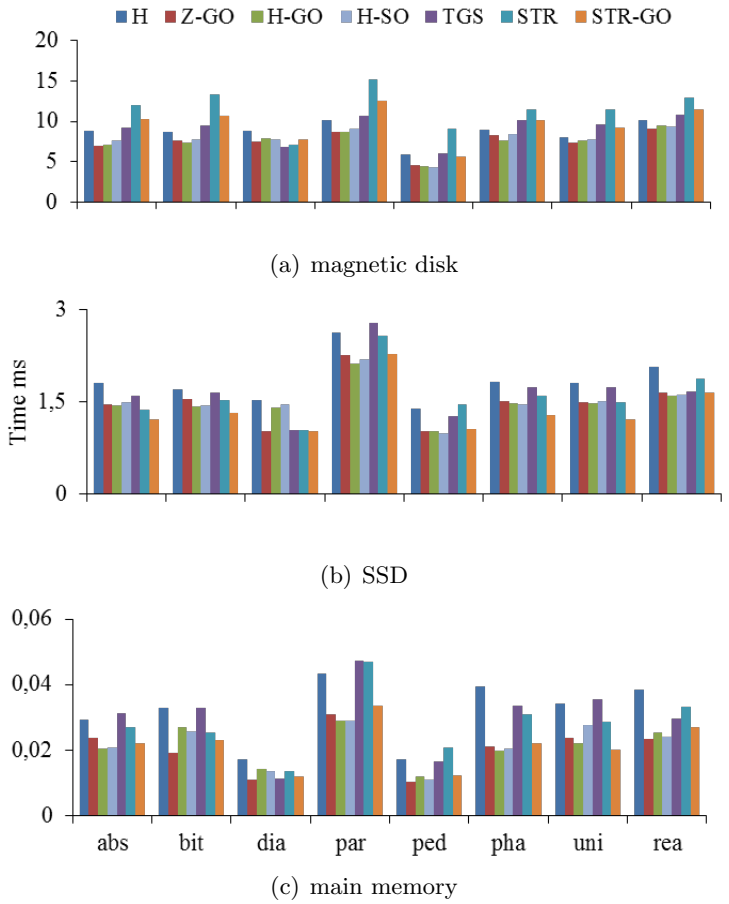


Figure 5.9: Avg. time per query for $qr2$ and d=2

In Fig. 6, the cost of our analytical model is plotted as a function of the number of leaf accesses required for processing the queries from profile $qr1$, $qr2$ and $qr3$ on the 2-dimensional data sets (both graph dimensions are normalized to H-loading). The graph shows a clear correlation between the cost measures . This supports that our cost model is indeed a good predictor for the actual cost. There are only three outliers corresponding to the extreme *dia* dataset, for which the real cost of the queries is substantially lower than the estimated costs model of our model.

Finally, the average total loading time of the algorithms for d=2 is depicted in Table 5.4. The cardinality of the data sets was limited to 1'000'000 rectangles. The total loading times of H/Z-loading, STR, H-GO, H-SO exhibit low standard deviation (see column std), while TGS is sensitive to the data distribution. H-GO loading time was clearly dominated by the time of external sort while the partitioning step itself has only little

	d=2	d=3	d=9	d=2,3,9
Z-GO	75.5 %	71.6 %	66.45 %	71.2 %
H-GO	76.2 %	73.3 %	68.3 %	72.6 %

Table 5.3: Avg. query performance of Z-GO and H-GO-loaded R-trees over square-shaped queries for different dimensionalities in leaf accesses (results are normalized to H-loaded R-trees)

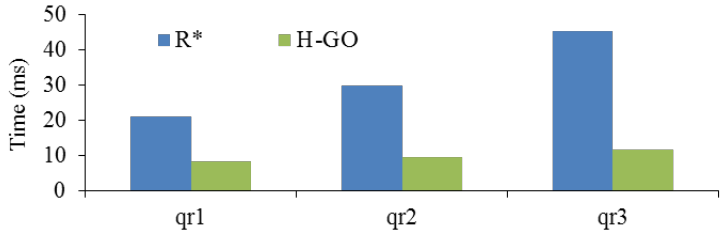


Figure 5.10: Avg. time per query for R*-tree and H-GO for California set (d=2)

impact (see build time). This differs from H-SO, where the time for the partitioning step dominates sorting, also STR is more expensive as data has to be sorted twice for d = 2.

5.5.4 Order Optimization

In this section, we primarily discuss the benefits of adaptive shuffling for better adaptivity to the underlying query profile. For the following discussion, we consider the results obtained from R-trees generated for the 2-dimensional uniformly distributed data set and query sets $qr2_a$, $a = 1, \ldots, 20$. Fig. 5.12 shows the average number of leaf accesses for $qr2_a$ queries as a function of the aspect ratio a. For each setting of a, we present the performance of five loading techniques ZAS, Z-GO, ZAS-GO, H-GO and H. Note that $a = 1$ represent the case of square-shaped queries. For $a = 1$ the performance of ZAS is identical to Z-loading. In agreement with previous experiments found in the literature, H-ordering is superior to Z-ordering. However, Z-GO and ZAS-GO are superior to H-loading and only slightly inferior to H-GO. For $a = 20$, the situation has changed dramatically. The performance of H-ordering has slightly decreased to 75% of Z-ordering, while ZAS-GO is clearly the most efficient technique. It is also evident from the comparison of ZAS, Z-GO and ZAS-GO that both of our techniques contribute to the substantial improvements that are observed for ZAS-GO. Moreover, GO in combination with Z-ordering provides slightly better results than H-ordering with GO.

The average volume V of leaf box is to be known in order to design our two-part space-filling curve. Assuming a uniform data distribution we can estimate the average

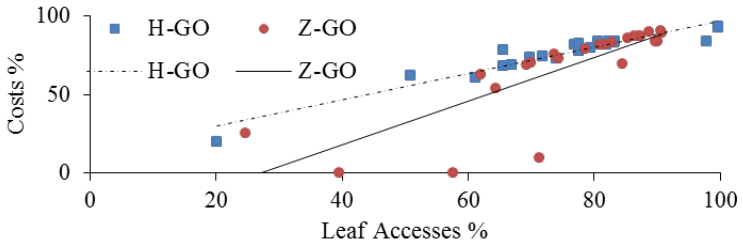


Figure 5.11: Correlation between the number of leaf accesses and the analytical costs

alg.	sort time	build time	total time	std
H	25,64	0.68	26.4	2.09
H-GO	25,64	7.40	33.12	2.03
H-SO	25,64	77.67	103.11	2.56
TGS	n/a	n/a	245.18	124.33
STR	n/a	n/a	55.47	7.76

Table 5.4: Avg. loading time (in sec.) of 1'000'000 2-dimensional rectangles.

leaf box volume by using the ratio of B and N. For $qr2_8$, the empirically determined optimal global value GR is compared with the estimated one. Our cost model returns $GR = 12$ for uniformly distributed data in all cases, which is in agreement with half of our experimental results (*abs, pha* and *uni*). The optimal value $GR = 14$ for *bit* slightly deviates from the estimated one. For *dia*, $GR = 0$ is the best value as the data records are located on the diagonal (it is sufficient to organize the data according to one of the axes). For *par*, *ped* and *rea*, the optimal value for GR was greater than 12 because the distributions are clearly non-uniform. In particular, data sets *par* and *ped* have a very high variance of volume and perimeter. This kind of data distributions is difficult to deal with and query adaptivity often yields no improvement.

To address this issue and verify our assumption, we used histograms as an option for deriving the values for sorting parameter GR. Histograms were also used for approximating dx_i value distributions for non-uniform data and query distributions. For each bucket p_i, our sort-based two step algorithm is processed independently with local parameters GR as well as average dx_i for p_i. Data summaries are held in memory and serve as a look-up function during the sorting and partitioning steps. We used the MinSkew-Histogram[10] with 100 buckets to represent the 2-dimensional data distribution. We observed that using histograms improves substantially the performance over global estimated parameter GR for $qr2_8$ profile (e.g. by 35% for *par*, by 15% for *ped* and by 46% for *rea*), while the degree of improvement depends not only on the chosen

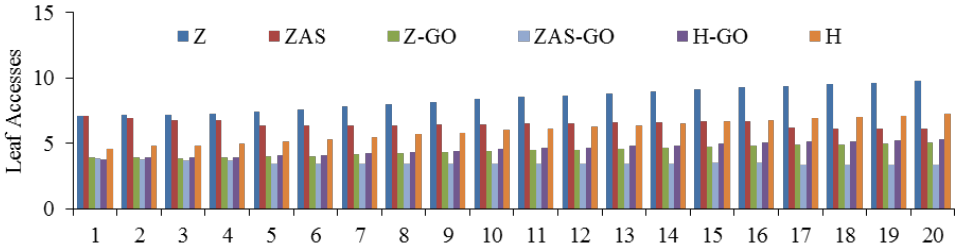


Figure 5.12: Query results for *uni* set (d=2)

histogram method but also on a histogram parameter settings. Therefore, we want to study more deeply the histogram and bulk-loading interaction in our future work.

5.5.5 R-tree for Intervals in Two-Dimensional Space

Here, we report experimental results with two-dimensional interval data sets. Our goal is to show that proposed framework improves performance of R-trees used for managing multiversion (partial persistent) data. As partial persistent records are mapped to intervals in two-dimensional space, they can be indexed using R-tree structure [105]. We map the end time stamp of live records to the maximal time stamp from an input file incremented by one. According to the convention, the time dimension is mapped to the X-axis and the key dimension to the Y-axis, respectively. This yields a set of two-dimensional intervals that are parallel to the X-axis.

In our experimental results presented in Chapter 4.7, we observed that MVBT+ gives substantially better query performance than an R-tree that is loaded using STR loading approach. In this section, we show that an R-tree loaded using the technique proposed for query adaptive loading combined with the adaptive Z-curve exhibit better query performance than R-tree loaded using STR algorithm.

In our experiments we used interval data sets obtained from the partial persistent files $u0$, $u50$, $u100$ used in MVBT experiments (see Chapter 4.7). For querying we used query files from the MVBT experiments. Query files consists of $10'000$ uniformly distributed rectangles. Each query rectangle have the same response set cardinality of 100 intervals. The aspect ratio is roughly 1:4. We used 8 KB pages for R-tree and MVBT in our experiments.

For indexing intervals we used the following steps. For generation a sorting order, we applied the adaptive Z-curve algorithm. In contrast to experiments previously reported in this section, we used the input intervals (a data set) instead of a query set for generating a sorting order. Both symmetric and asymmetric part of a Z-string starts with

Y-axis (key-dimension), as the key dimension has no extension. We used the average length of time interval (X-axis extension) for generating GR parameter (the length of symmetric prefix). We experimented with different sorting orders. We noticed that the best results are obtained using lexicographical ordering on the key and time dimensions (Y-axis and then X-axis) for this data sets. As there is at least 10% of the data is live, the average interval length is also very large. This also yields lexicographical order using our adaptive Z-Curve approach. The impact of interval length and Z-curve orientation on the indexing interval data requires more detailed investigation.

After sorting the data we applied our *gopt** partitioning optimizing the volume. We summarize our used approach for indexing static set of intervals in two-dimensional space as follows:

1. We compute an average interval length (time duration) from a data set. We use AdaptiveShuffle algorithm to generate a sorting order. The input of AdaptiveShuffle is always a pair Y-axis (key-dimension) and X-Axis (time-dimension). The length of symmetric prefix is computed based on the average interval length.

2. We sort intervals using sorting order obtained from step 1.

3. We bulk load R-tree using *gopt** partitioning method.

For sorting in step 2. we use start point of the interval.

Table 5.5 displays results of query experiments. MVBT$^+$ is weight balanced MVBT loaded using our new loading procedure. MVBT-LRU is standard MVBT loaded using tuple by tuple method. R-TREE is a sort based loaded R-tree using lexicographic ordering on key and time dimension. R-TREE GOPT is a sort based loaded R-tree using our *gopt** optimized approach (also lexicographic ordering on key and time dimension). H is an R-tree loaded using Hilbert curve. H-GO is *gopt** optimized R-tree loaded using Hibert curve.

According to our preliminary results we observe that well tuned R-trees exhibit a good average I/O performance for interval data. R-trees loaded using lexicographical order display at most roughly a factor of two more I/Os in comparison to MVBT. For a data set $u0$, optimized R-TREE GOPT requires less average I/Os per query and almost the same number of average leaf accesses. The height difference is due to the smaller number of live records in nodes of MVBT-LRU and MVBT$^+$ in comparison to the page capacity of the R-tree. Additionally, both MVBT-LRU and MVBT$^+$ need to access root* for historical queries. In general, we observed greater influence of sorting order on query performance than the optimal partitioning. Other orderings than the lexicographical

order were inferior. We observed that R-trees loaded with Hilbert o exhibit a high MBR overlap.

Workload file		$u0$	$u50$	$u100$
MVBT$^+$	I/O	4.69	4.75	4.8
	leafs	1.72	1.78	1.83
MVBT-LRU	I/O	4.3	4.18	3.83
	leafs	1.72	1.79	1.82
R-TREE	I/O	4.97	5.75	8.94
	leafs	1.97	2.75	5.91
R-TREE GOPT	I/O	3.79	4.41	6.93
	leafs	1.79	2.4	4.94
H	I/O	108.33	144.39	128.51
	leafs	96.55	128.91	112.6
H-GO	I/O	74.15	102.76	95.1
	leafs	64.5	90.47	82.42

Table 5.5: I/Os and leaf accesses for two-dimensional interval data

Our preliminary results, are very promising. R-trees offer a similar query performance like a MVBT if query-adaptive loading techniques are used for their generation, but the loading of R-trees is substantially faster.

5.6 Conclusions

In this work, we reconsidered the problem of sort-based bulk-loading of R-trees. We demonstrate the importance of query profiles for search efficiency of generated R-trees. We designed new loading algorithms based on two innovative techniques. The first consists of a new sorting technique of rectangles based on non-symmetric Z-order curve design, while the second generates an optimal partitioning for a given sequence of rectangles. Both techniques are optimized according to a commonly used cost model for range queries. Our optimal partitioning techniques are broadly applicable and beneficial. They can be easily integrated into other loading techniques like STR, which is a popular loading method in commercial database systems. They can also be combined with standard Hilbert-loading even when the query profile is unknown. In this case, we suggest to use the partitioning that minimizes the area of the bounding boxes of the leaves.

Our experimental results obtained from a standardized test framework clearly reveal the advantages of our techniques in comparison to standard loading techniques (STR, Hilbert-loading, Z-loading, TGS). Our techniques creates R-trees with consistently better

search efficiency than those created by pure Hilbert-loading, while for some data files large improvements in query performance (about factor 5) were achieved. Interestingly, due to our new partitioning methods, there is no noticeable differences anymore in the performance of R-trees build form rectangles sequences following either Hilbert-ordering or Z-ordering. Thus, we suggest using Z-ordering because of its conceptual simplicity.

6 Construction of R-tree-Based Histograms

In this chapter, we present methods for spatial histograms construction using the query adaptive partitioning framework presented in Chapter 5. Spatial histograms are integral part of an efficient spatial query processing and are used for the result estimating of spatial queries. We show that the task of generating spatial histogram of a high quality is very similar to generating R-trees with a good query performance. Both of them rely on partitioning of a set of rectangles into disjoint subsets. The computation of the optimal spatial histograms is a non-trivial task and in general it is NP-hard [90]. Therefore, we investigated heuristic methods for fast histogram construction. Our major research goals are:

1. A high estimation accuracy of resulting histograms.
2. The low I/O and CPU costs of histogram construction.
3. Robustness for different query and data distribution.
4. Avoiding performance sensitive parameters.

The groundwork of our histogram construction methods is a query adaptive loading algorithm for R-trees (see Chapter 5). We generate spatial histograms in the I/O complexity of external sort simultaneously with a bulk-loading of R-trees. Major parts of this chapter were published in [4, 5].

6.1 Introduction

Histograms are important data structures primarily used in database systems for estimating the selectivity of queries. They are also applied to obtaining quick approximate response for aggregate queries. While one-dimensional histograms are widely available in almost all database system, only a very few systems offer multidimensional histograms. Most of them are simple grid-based methods that are applicable to two-dimensional point data only. These methods perform poorly on rectangle data or when the independence assumption of the attributes is violated.

The design of efficient multidimensional histograms turns out to be much harder already for the two-dimensional case. In fact, the problem of designing optimal multidimensional histograms is known to be NP-hard. Therefore, many heuristics have been developed and evaluated in various experimental settings. In general, these heuristics result in fairly complex parameterized algorithms with a runtime often substantially higher than the runtime of the one-dimensional counterparts. This is often not acceptable because histograms have to be rebuilt quite frequently. In addition, the algorithms are often quite sensitive to small variances of the parameter values.

We revisit the problem of designing efficient multidimensional histograms from the perspective of bulk-loading spatial index-structures, e.g., R-trees. Similar to R-trees, a histogram is viewed as a set of bounding boxes, but each of them is associated with statistical information e.g. the number of spatial objects that are assigned to the box. Rather than directly generating histogram buckets, our method relies on a two-step approach: First, the leaf level of an R-tree is generated and second, adjacent leaves are merged into larger histogram buckets. Crucial and sensitive parameters are avoided; instead both steps rely on the optimization of a widely accepted cost function. This makes our approach very appealing to an end-user.

Even though our optimization bases on minimizing a cost function, it still remains a heuristics like it is for all other multidimensional histograms. It is therefore of utmost importance to use a thoroughly designed experimental setup to provide a meaningful and fair comparison with competitors. So far, there is no commonly agreed experimental setup for spatial and multidimensional histograms. In particular, we found serve deficiencies in current experimental work, e.g., small data sets, low selectivity of queries, uniformly distributed queries.

Our contributions are summarized as follows:

1. We present a uniform rectangle partitioning framework for R-tree loading and histogram construction. Derived from this framework, we present an efficient two-step approach to generating multidimensional histograms.

2. We introduce query models for workload generation and examine the accuracy of histograms under these workloads.

3. We present an experimental performance comparison of a large number of multidimensional histograms.

6.2 Preliminaries

In this chapter, we investigate the problem of R-tree based histogram construction for a d-dimensional set of N rectangles $\{r_1, \ldots, r_N\}$. The groundwork of our method is the sort-based bulk loading algorithm for R-tree presented in Chapter 5. We assume that R-trees have the node capacity B and the minimum occupation $b \leq \lceil B/2 \rceil$. Our description will address the case $d = 2$; the generalization for $d > 2$ is only discussed when necessary. In addition to notations introduced in Table 5.1 Chapter 5, we summarize notations used in this chapter in Table 6.1.

We define the output histogram H as a set of buckets $h_1, \ldots, h_m$. The bucket h_i contains statistical information about the set of spatial objects (rectangles) $R_i = r_1, \ldots, r_n$. These are [10]: $MBR(h_i)$ of set R_i, number of elements n_i, average rectangle side length dx^i_{avg}, dy^i_{avg} over set R_i and spatial density information $s_i = Area(h_i)/Area(MBR_{h_i})$. Where, $Area(h_i)$ is defined as a sum of rectangle areas in R_i and $Area(MBR_{h_i})$ is defined as an area of bucket MBR.

Hereafter we use notion bucket and MBR as synonyms depending on the context. Further, our goal is to build the histogram in such way that the number of elements referenced in each bucket varies only by a small constant factor, so that H is close to an equi-depth histogram. Both spatial histogram buckets and R-tree nodes are associated with a disjoint subsets of an input rectangle set. Hence, R-tree can be extended to a spatial histogram by attributing additional statistical information [10, 71, 53, 21] to R-tree nodes.

The selectivity estimation $est(q)$ for range q_w and point queries q_p is computed based on the uniform distributions assumption [10]. The selectivity estimation of a point query is computed as follows: Let MBR_{h_i} be the bucket MBR containing a query point q_p. Then s_i is an average number of rectangles hit by given point query in the bucket h_i.

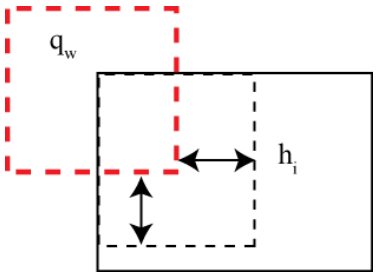


Figure 6.1: Range query estimation.

Consider a range query q_w. Let MBR_{h_i} be a bucket MBR overlapping with the query q_w. Let $r_s = MBR_{h_i} \cap WQ_{q,s}$ be an intersection rectangle. r_s is represented by

its center (cx_{r_s}, cy_{r_s}) and its extensions (dx_{r_s}, dy_{r_s}). We extend then (dx_{r_s}, dy_{r_s}) with $2dx^i_{avg}, 2dy^i_{avg}$ in both dimensions with a constraint that the extended sides cannot cross the boundaries of MBR_{h_i} (see Figure 6.1). Then

$$est(h_i, q_w) = n_i \cdot \frac{Area(r_s)}{Area(MBR_{h_i})} \tag{6.1}$$

is the estimated number of rectangles intersecting q_w for a single bucket h_i [10]. By this, the overall estimation for q_w is defined by the sum of $est(h_i, q_w)$:

$$est(q_w) = \sum_{i=1}^{m} est(h_i, q_w) \tag{6.2}$$

Similar to R-trees, the average estimation error depends on the number of rectangles that overlap the query q_w and the rate of uniform distribution of objects in the buckets.

The histogram space consumption is a crucial design aspect, as histograms reside in main memory. We consider the space consumption in terms of allocated machine words. Let w be a size of machine word in bytes. Each bucket h_i manages MBR and the following statistical values: $n_i, dx^i_{avg}, dy^i_{avg}, s_i$. Thus, storage amount of a bucket h_i is

$$d \cdot w + 2 \cdot w + d \cdot 2 \cdot w$$

where $d \cdot w$ bytes are average length information, $2 \cdot w$ bytes are used for number of objects and spatial density and $d \cdot 2 \cdot w$ bytes to represent MBR of h_i (k is the number of dimensions). This storage scheme is also used in MinSkew approach [10].

Our goal is to achieve a compact representation of a data set with a high estimation quality for a given space budget. The motivation for our for heuristic method is by the results presented in [90]. In order to quantify and predict the estimation accuracy of histogram H, we introduce weight function $f_w(h_i)$ that returns a numerical value for a bucket. We define the cost of a histogram H as the sum of $f_w(h_i)$ weights. The space budget in terms of machine words yields the number of allowed number of buckets m for histogram H.

Definition 3. *For a given space budget m in terms of number of buckets our task is to build a histogram H such that*

$$C_h = \sum_{i=1}^{m} f_w(h_i)$$

is minimized for a given rectangle set.

notations	description
$H = h_1, \ldots, h_m$	a spatial histogram obtained from a set of input rectangles.
h_i	a histogram bucket. Each bucket h_i refers to a disjoint subset of input rectangles. Additionally, to each bucket is a statistical information obtained from this subset attributed.
R_i	A disjoint subset of input rectangles referred by bucket h_i
$n_i = \lvert R_i \rvert$	Number of rectangles in R_i.
$MBR(h_i)$	minimal bounding box (rectangle) of bucket h_i, computed over set R_i.
$Area(h_i)$	sum of rectangles areas is R_i.
$Area(MBR(h_i))$	area of $MBR(h_i)$.
dx^i_{avg}, dy^i_{avg}	average side lengths of rectangles in set R_i.
$s_i = Area(h_i)/Area(MBR(h_i))$	spatial density of bucket h_i.
$f_w : h_i \to \mathbb{R}^+$	weight function for buckets h_i
$C_h(H) = \sum_{i=1}^{m} f_h(h_i)$	cost function of histogram H

Table 6.1: Important notations

Intuitively, we want to quantify the spatial distribution within a bucket and choose histogram H with a lowest sum of weights.

6.3 Related Work

During the last three decades one-dimensional histograms have been used widely for the purpose of selectivity estimation and with a fair amount of success [98, 72]. Nevertheless,

in the case of multidimensional histograms, we are still facing many challenges, that need to be solved [71]. To tackle these problems, many different heuristic based methods were proposed. All of them aim to partition the multi-dimensional data in rectangular buckets for a given space budget. The data within buckets is uniformly distributed, since the query estimation relies on uniform data distribution assumption. The heuristic methods are motivated by the results in [90]. The authors show that computing the non-overlapping rectangular partitioning with near-uniform data distribution within buckets is NP-hard [90, 71, 53, 103].

One of the first methods proposed for multidimensional data is *hTree* [88]. It constructs non-overlapping partitioning of multidimensional space based on a frequency as source parameter. Only one dimension is approached at a time and partitioned in buckets with an identical number of objects, resulting in a *equi-depth* histogram [88, 53]. The advantage of hTree is its low construction cost. However, the partitioning rule is too rigid for highly skewed data [88, 103]. In contrast, *mHist* uses space partitioning [100]. Space is partitioned along the dimension that benefits most from a split. The split decision is made based on a marginal frequency distribution [100, 71, 53]. This approach was developed for relational data and focuses mainly on approximating point frequencies. However, selectivity estimation for spatial data differs from traditional one [10]. The object frequencies may be uniform, but the locations can be highly skewed, and the objects vary in sizes and shapes.

To provide accurate estimation for spatial objects and also I/O efficiency, the *MinSkew*-Histogram method was proposed [10]. The authors proposed two construction strategies. The basic variant works as follows: in the first phase, the algorithm computes a regular grid and stores the number of intersecting spatial objects for each cell. Based on the computed grid, the recursive binary space partitioning (BSP) is used for histogram computation. The buckets are picked for further processing based on a split value that will lead to greatest reduction of data skew. The decision is local, so that for all dimensions, all possible cuts based on marginal objects frequencies are considered. The authors observed that a fixed grid size is sensitive to the size of queries [10] (high grid resolution favors small sized queries and small resolution large queries). To lessen this effect the second construction strategy *MinSkew-Progressive-Refinement* utilizes grids with different resolutions. Each grid resolution is used to construct the equal portion of histogram buckets. The computation is processed in top-down fashion starting with a low resolution grid applying BSP in each step. The downside of both strategies is that the performance is sensitive to the grid resolutions.

GenHist proposed by [64] tries to identify high density regions. In contrast to the

previous methods, the bucket rectangles may overlap. Moreover, the buckets can be contained in other buckets. GenHist finds regions with high object density, excises them but leaves enough data in the parent bucket so that the parent buckets distribution flattens. Again, the method uses a regular grid as a starting point for histogram construction.

The recently proposed method *STHist* [103] applies the idea of *GenHist* to 2-3-dimensional spatial objects. In the basic variant decision about whether the region is dense is made by applying a sliding window over all dimensions, approximating the frequency distribution by a marginal distribution. The dense regions called *Hot-Spots* build hierarchies, so that the Histogram is represented as an unbalanced R-tree. In the advanced variant called *STForest*, the algorithm first computes coarse partitions according to the object skew, and then applies a sliding window algorithm to them. The idea behind this is that if the region is already uniformly distributed further partitioning is unnecessary. Moreover, the coarse regions merge together if the skew of merged bucket decrease. The experiments conducted in [103] show that STHist is superior to other proposed methods. However, STHist has time complexity $O(n^2)$ for 2-dimensional and $O(n^3)$ for 3-dimensional data.

Recently, the class of self-tuning histograms like *STHoles* and *ISOMER* were proposed [47, 111]. In general these methods incrementally update buckets and their frequency information, using query feedback. These kind of methods is very appealing, because the incremental modification the histogram adapts to the real distribution of a data. Moreover, the methods can be applied independently on top of different approaches.

Another way to obtain a spatial histogram is to generate it using a spatial index structure like R-tree [10, 21, 71, 53, 33]. Paul Aoki in his work [21] introduced a generic approach for selectivity estimation for a wide range of tree base index structures [20]. He proposed to combine tree based index and histogram in one structure. His work is inspired by previous work of Antoshenkov [19] where B^+trees is used to obtain a random sample. For this purpose, tree nodes manage additional information about the leaf node cardinalities. The cardinality of the internal node is defined by a sum of child cardinalities. The managed information allows to compute the upper and lower bounds. Aoki designed top-down traversal algorithm that uses this information for the selectivity estimation [21]. In his experiments, he reported that bulk loaded R-trees exhibit a high estimation accuracy. In contrast to our work, in [20] the standard sort based loading technique such as STR [80] method was used. In our work we show that this yields not always a good partitioning.

The recently proposed approach *rKHist* [53] is also based on R-tree bulk-loading procedure [73]. The data is presorted according the Hilbert space-filling-curve. After the leaf nodes are generated, one possibility to generate a histogram is to pack nodes according

to the sorting order in equi-sized histogram buckets. This leads not always to a good partitioning. Especially, for near-uniformly and uniformly distributed data equi-sized partitioning wastes buckets for regions with a high object density and yield high overlap, despite the fact that the regions have uniform distribution [10]. Therefore, the authors proposed a greedy algorithm that utilizes a sliding window of pages along the Hilbert order. The algorithm is parametrized with a number of buckets that should be considered for a splitting. A bucket-split is applied if it leads to an improvement according to the proposed cost function.

Our approach differs from rKHist in that we tune the R-trees according to the widely used R-tree cost model. Our generic sort-partition framework computes optimal partitioning for a given cost function according to the sorting order of rectangles. The framework relies on the dynamic programming scheme proposed by [72] for generating one dimensional V-optimal histograms.

A recent approach [33] uses also R-tree for spatial histogram generation. The initial histogram bucket is obtained from a predefined leaf node set. Their basic variant reads predefined number of leaf nodes as well as stored object geometries using depth-first top-down traversal. Their general idea is a binary recursive partitioning of the initial bucket. They apply equi-count as well as equi-area partitioning rules. An equi-area heuristic splits a bucket along the dimension with longest side in two equal area buckets. Then it distributes spatial object using their centers among the new buckets. Large object geometries that overlap both buckets are attached to a both buckets. They update statistical information stored in a bucket such as average extents and number of objects while splitting a bucket. Similarly, equi-count rule splits bucket based along the dimension with the maximal number of distinct projected center values. The bucket with a most gain according to the split rule is chosen at each step. Additionally, only buckets with a predefined number of elements are considered for a split. The each split need to access actual object geometries, this yields high I/O overhead. Therefore, authors proposed to use for the initial bucket an MBR of a sub tree such that all referenced geometries can be processed in memory. Our approach differs from [33] that we rely on R-tree bulk loading procedure. Moreover, even for existing R-tree our new approach requires only one pass through the input data.

The scheme proposed in [72] is also used in [120] for computing a set of k minimal bounding rectangles (MBR) from a 2-dimensional point set. The goal was to reduce communication costs for mobile devices by approximating the spatial query result by a set of MBRs with a minimal information loss f_i. The authors showed that computing such representations is NP-hard even for d=2. One of their heuristics first sorts the

query output using the Hilbert order and then apply the partitioning method of [72]. Multi-dimensional histograms and representation with a minimal information loss f_i are related, since both techniques are considered as data summarization methods. In contrast to histograms the optimization function is different and space constraints are disregarded.

In this work, we adapt the dynamic programming scheme [72] for a R-tree based histogram generation introducing the space constraints on bucket capacity size. This allows us to generalize the partitioning scheme and to design new more efficient algorithms for R-tree and R-tree based histogram generation. We show that especially for highly skewed data, the R-tree methods return more accurate results. Moreover, R-tree histograms constructed using our dynamic programming framework display good estimation accuracy for near-uniform and uniform data sets.

6.4 R-tree Framework

In order to obtain a high-quality histogram H, the data should be partitioned in such way that the data within each histogram bucket is near-uniformly distributed. Computing such partitionings is a non-trivial task and in general NP-hard [90]. Furthermore, the way how data is partitioned also influences the quality of the R-tree (see Chapter 5). Partitionings minimizing sum of MBR volumes yield better R-trees according to the cost model [73, 95, 115]. In order to obtain a partitioning in polynomial time, we use a heuristic method based on SFC. Our approach can be summarized as follows: reduce the complexity of multidimensional partitioning by sorting the data according to a SFC, and solve the partitioning problem optimally for the sorted set. In the following, we present a high level description of the building blocks of our framework:

- **Sort-Partitioning:** Sort the rectangles with respect to a SFC. Partition the sorted sequence optimally according to a cost function into subsequences of size between b and B.
- **Bulk-Loading R-tree:**
  - *Step 1. Node Generation:* Run *Sort-Partitioning* with parameter settings for b and B according to a given page size.
  - *Step 2. Generation of Index Entries:* For each page, compute the bounding box of its partitions and create the corresponding index entry.

  - *Step 3. Recursion:* If the total number of index entries is less than B, store them in a newly allocated root. Otherwise, start the algorithm with the index entries (bounding boxes) from Step 2.

- **Construction of Histogram H:** Run step *Node Generation* of bulk-loading. Collect and store statistics. Run *Sort-Partitioning* on generated leaf nodes.

Sort-Partitioning is the crucial step in the algorithm for bulk-loading indexes and generating histograms. The first step uses SFC to sort the data. If query profile is unknown we consider Hilbert ordering otherwise we apply adaptive Z-Curve (see Chapter 5). The second step partitions the sorted sequence of rectangles in optimal way according to a cost function. Note that our approach is a heuristic and rely on the specific sorting order. The difference between loading indexes and generating histograms is that histograms do not require a recursive processing.

Sort-Partitioning uses partitioning algorithms presented in Chapter 5. They are:

1. Query-optimal partitioning:

$$gopt^*(i) = \min_{b \leq j \leq B} \{gopt^*(i-j) + f_w(p_{i-j+1,i})\}$$

2. Storage-Bounded partitioning:

$$opt^*(i,k) = \min_{b \leq j \leq B} \{opt^*(i-j,k-1) + f_w(p_{i-j+1,i})\}$$

Since the $gopt^*(i)$ has lower CPU and I/O costs than $opt^*(i,k)$ as well as it produces better R-trees, we apply this partitioning scheme in Step 1. (Node Generation) and Step 2. (Generation of Index Entries). As a weight function for Query-optimal partitioning we use V (volume of MBR) and if applicable V^+ (extended volume of MBR obtained from query profile).

Storage-Bounded partitioning is applied in Step 3. (Construction of Histogram H), as our goal to build a histogram for a given space budget. Our framework allows us to choose weight function f_w to quantify the spatial distribution within bucket h_i. There are four factors that contribute to the CPU costs of $opt^*(i,k)$ partitioning, they are:

1. The number of input rectangles.
2. The maximal number of rectangles per bucket.
3. The target number of histogram buckets m.

4. The computation costs of weight function f_w.

By this, we consider weight functions with low CPU costs. In Chapter 5 we used $f_w := V(MBR(p_{i-j+1,i}))$ and $f_w := V^+(MBR(p_{i,j}), QP)$, both functions are computed in $O(1)$ (more precise $O(d)$) for a given MBR. Similarly, for a histogram construction, we consider volume of MBR (for $d = 2$ area) as a default weight function $f_w =: V(MBR(p_{i-j+1,i}))$. The combination of sorting the data according to SFC and $opt^*(i,k)$ partitioning with $V(MBR(p_{i-j+1,i}))$ as a weight function yields a tight representation of an input set consisting of m buckets. Intuitively, the partitioning with the lowest sum of volumes influence the spatial distribution of input rectangles in histogram buckets, as the objects are more spatially clustered within them.

6.4.1 R-tree Histogram

Here, we discuss the histogram construction in detail. We construct histogram H using our framework as follows:

1. **Micro-Clustering Step:** (we use the same terminology as in [14]) First, the leaf pages of an R-tree are generated using partitioning $gopt^*$. The following parameters needs to be set: space-filling curve, bucket capacity parameters b and B, and weight function f_w. As a default space-filling curve, we use the Hilbert curve. The parameters B and b of the initial step are adjusted to the system physical page size. The default weight function is V (minimizing the area of the bounding boxes). For each leaf, also termed micro-cluster, we compute the required statistical information (number of objects per leaf, average side lengths dx, dy and the density s). Let m_1 denote the number of leaves.

2. **Histogram-Generation Step:** Given the number m of required buckets and assume $m < m_1$, we apply opt^* to the bounding boxes of the leaf pages generated in the micro-clustering step. Thereby, we obtain the final m buckets of histogram H. The minimal occupation b_2 and the capacity B_2 are set in the following way: $b_2 = \max(\lfloor m_1/2m \rfloor, 1)$, $B_2 = \lceil m_1/m \rceil + b_2$.

   Finally, we compute the statistical information (number of objects, averages side lengths, density) by aggregating the statistical information in the associated leaves. As a weight function we use V (minimizing the area of the bounding boxes).

Although the first step could be skipped, it is important to use it for the following reasons: Firstly, it reduces the time complexity of the final histogram construction.

Secondly, the histogram is generated simultaneously with R-tree index. Thirdly, it can be implemented within the same bulk loading routine.

Let us discuss the total CPU and I/O costs of our two-step method in more detail. The I/O costs for sorting the input set is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. Micro-clustering step is computed using $O(N/B)$ I/Os as well as histogram-generation step. Thus, the total I/O costs are bounded by the costs of external sorting.

The CPU cost of the micro-clustering step is $O(N \cdot B)$ (see Chapter 5.3). For now we assume that m_1 buckets corresponding to m_1 leaves generated in Step 1. can be hold in memory. The CPU cost of the histogram generation step is equal to $O(m_1 \cdot B_2 \cdot m)$. Because $B_2 = O(m_1/m)$, it follows that the final cost is $O(m_1^2)$. From $m_1 = O(N/B)$, we obtain $O((N/B)^2)$ CPU costs for Step 2. The direct application of the Histogram-generation step would be $O(N^2)$, a factor of B^2 more expensive than the two-step approach. According to our experimental results, this method shows high accuracy and robustness of selectivity estimation for different data and query distributions [4]. If there is not enough memory to apply storage-bounded partitioning, we first generate chunks of equal size and apply opt^* to every chunk. Similar strategy we used for generating query adaptive R-trees in Chapter 5.

In the histogram generation step we use $opt^*(i,k)$ partitioning to find a best partitioning according to the sum of MBR volumes. In general, the bounded partitioning scheme allows us to use other weight functions. However, our experimental results show that histogram built using this weight function displays high accuracy and is computed with low CPU overhead. $opt^*(i,k)$ partitioning scheme can be adapted also for other cost function than sum of weights e.g. if we consider the cost of a histogram H by a maximal weight $C_{max}(H) := \max_{i=1}^{m} f_w(h_i)$ we replace the summation with the maximum computation. However, experimental results show that combination of C_{max} and volume of MBR produces histograms with an inferior estimation accuracy than default cost function according to used error metrics. We let the investigation of other cost and weight functions and their performance for different error metrics for a future work.

6.5 Experiments

In this section, we present summarized results obtained from a set of experiments under different query workloads. First, we describe the underlying query models, and then we provide details about our data sets and query files. Finally, we present a detailed discussion of the results.

6.5.1 Query Models

For experimental settings we followed a methodology for generating workloads based on query models originally proposed in [95] for the design of multidimensional index structures. The authors classified range queries according to the indicators *aspect ratio*, *location* and *size*. The query size is defined by either area (relative to the entire data space) or the number of qualified objects. Query location can follow either a uniform distribution or the distribution of the underlying data. The aspect ratio equals the width-to-height ratio of the query rectangle, which we assume to be 1 (quadratic windows) in the following. This yields in four different query models:

- M_1: size = area, location = uniform distribution,
- M_2: size = area, location = data distribution,
- M_3: size = number of answers, location = uniform distribution,
- M_4: size = number of answers, location = data distribution.

Data and Query Sets

For our experiments we use 2- and 3-dimensional data sets originally developed for RR*-tree evaluation [39] and used in experiments with query adaptive loading of R-trees. Detailed description of data sets is given in Chapter 5.5.1.

According to query models $M_1, \ldots, M_4$, we generated two workloads for each data set and each query model. Two query sets are generated from model M_1. The first one consists of $10'000$ uniformly distributed quadratic query rectangles with average volume $V = 0.01\%$ (so that under uniform distribution approximately 100 objects qualify for a data set with $1'000'000$ objects). The side length of the rectangles are uniformly distributed in range $\left[\frac{1}{2}V^{1/d}, \frac{3}{2}V^{1/d}\right]$. The second query set is generated in the same way with an average volume of 0.1% and consists of $3'164$ query rectangles.

The location of the queries from model M_2 follow the underlying data distribution. Again the average volume of the query sets were set to 0.01% and 0.1%, respectively. For the production of queries of model M_3 we first generated uniformaly distributed points and used them for issuing k-nearest neighbor queries with the maximum norm L_∞. The bounding boxes of these k-NN queries, k = 100 and k = $1'000$, are used for two sets of window queries with 100 and $1'000$ answers per query, respectively. For model M_4, we used the underlying data distribution for producing the reference points for the nearest neighbor queries. Thus, the location of the window queries also follows the data distribution. Again two query sets are generated with 100 and $1'000$ answers per query.

6.5.2 Studied Methods

In our experiments, we study the performance of different histograms. As a reference method we used MinSkew. Histograms produced by MinSkew perform well [64, 10, 103]. We implemented both MinSkew with fixed grid and *progressive refinement* strategy respectively, as described in [10]. We refer to the first as MinSkew and the second as MinSkewProg. For each data and query set we always used the best parameter setting for the grid size. For d=2, we used a grid with 2^{14} cells. This was the the best setting according to accuracy and build-up time in our experiments. For a MinSkewProg we used four grids with $2^{14}, 2^{12}, 2^{10}, 2^8$ cells; again this was the best setting. For d=3, we used 2^{15} cells for MinSkew and four grids with $2^{15}, 2^{12}, 2^9, 2^6$ cells for MinSkewProg. Other examined methods are listed in Table 6.2.

Histograms	Description
MinSkew	minSkew, fixed grid
MinSkewProg	minSkew, prog. refinement
rkHist	rK-Hist with $\alpha = 0.1$
R-tree	fixed sized partitioning, Hilbert Curve
R-V	V, Hilbert Curve
R-VQP	V^+, Hilbert Curve
R-RK	RK, Hilbert Curve
R-SK	SK, Hilbert Curve
FST	STHist forest

Table 6.2: Studied Methods

R-tree Methods

Methods with prefix R (R-V, R-VQP, R-RK, R-SK) are derived from R-trees and our Sort-Partition algorithms. For R-tree methods we set $B = 100$ and $b = 40$ for d = 2 and $B = 72$ and $b = 28$ for d = 3. Recall that B denotes the leaf capacity and b the minimum leaf occupation. Leaf nodes are generated using *gopt** algorithm. In general, this results in more buckets than m (the desired number of buckets). In a second step, we apply *opt** to the leaf bounding boxes to yield exactly m buckets. The chunk size was set to $20'000$ rectangles; larger chunk sizes did not yield significantly better histograms. The methods R-V, R-VQP, R-RK, R-SK only differ in their weight function used in *gopt** and *opt** (see Table 6.3). For example, V refers to the cost function minimizing the volume of the bounding boxes. Additionally, we implemented rKHist as described in [53] using an underflow rate with $\alpha = 0.1$ (again this was the best setting in our experiments).

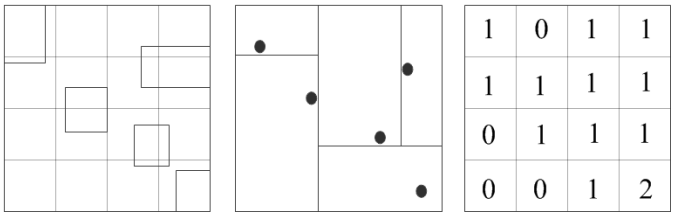


Figure 6.2: k-Uniformity metric and spatial skew of MBR

We also studied the quality of other weight functions (R-RK, R-SK). RK is a k-Uniformity metric proposed in [53]. SK minimizes the skew within a bucket. For a detailed description, see [53, 10]. Figure 6.2 illustrates how these functions are computed. The left figure shows a bucket region with five rectangles.

Weight Func.	Description
V	volume of MBR
V^+	V extended by avg. query side lengths
RK	k-Uniformity metric
SK	spatial skew of MBR

Table 6.3: Studied Methods

The k-Uniformity function RK is based on a rectangular subdivision of a bucket region. The last is built using the associated point objects [53]. For rectangles we only considered their centers. This subdivision is computed in kd-tree manner. This representation is constructed using the recursive binary splits of a point set. Each dimension of the rectangular bucket split into two, in a round-robin fashion. The median is used as the spit point, see center plot of Figure 6.2. The RK returns the standard deviation of areas of the resulting rectangles. Note that, the processing cost for a bucket with n elements is $O(n \log n)$. This is a drawback in comparison to other cost functions discussed above.

The function SK is based on a regular grid [10]. First, the regular grid is computed for a bucket region (see left side of Figure 6.2). Then, the frequency of objects intersecting a cell is computed for each cell (see right side of Figure 6.2). The function then returns the standard squared error (SSE) of frequencies. The drawback of the last method is that the grid resolution has to be set as an additional parameter.

We also implemented STHist method [103]. We call it FST, because we used the so-called *forest*-strategy. The build-up cost for FST is very high, particularly for data sets with skewed data distributions. For example, the build-up time was a factor 100 higher for the *rea* data set than for other methods, due to its $O(n^2)$ runtime. In order to conduct all the experiments for FST, we applied FST to a random sample of 10%.

Space Allocation

Recall that each bucket h_i maintains the MBR and the following statistical values: $n_i, dx^i_{avg}, dy^i_{avg}, s_i$. Let w be a size of machine word in bytes. Thus, storage amount of a bucket is $d \cdot w + 2 \cdot w + d \cdot 2 \cdot w$ ($d \cdot w$ bytes are average length information, $2 \cdot w$ byte for number of objects and spatial density and $d \cdot 2 \cdot w$ for MBR). It is possible to save storage for MinSkew bucket MBR. Because the MinSkew histogram can be stored as a kd-tree. We assume that the leaf node of kd-tree stores statistical information. We implemented MinSkew using a grid with resolution of power 2. For 2^{dk} cells we need $\log k$ bits to decode the split position. Additionally we store information about the split dimension $\lceil \log d \rceil$ bits and one bit to decode whether a node is a leaf [56].

Thus, for d=2 and d=3, the MinSkew Histogram can keep almost twice as many buckets as R-tree histograms. This is reflected in our experiments. If an R-tree histogram consists of m buckets, we allow MinSkew to use $2 \cdot m$ buckets. In our experiments, space allocation is expressed by the number of buckets m for R-tree Histogram.

All methods are implemented within the XXL-library [42]. The experiments are conducted with a 64 bit Intel i7-2600 (2 x 3.4 Ghz), 8 Gb memory machine running Windows 7. For external sort we used 10 MB memory buffer. We examined histograms with $m = 500, 1000, 2000, 3000, 4000$ and $m = 5000$ buckets. Note that previous experiments considered only a small number of buckets. Due to large main memories available, we see the necessity to investigate large histograms.

Error Metrics

Performance quality of the proposed methods was evaluated using different error metrics. We use workload error E_w as default metric. E_w is defined as follows:

$$E_w = \sum_i |act_i - est_i| \,/\, \sum_i act_i$$

Here, act_i is the actual number of answers of the i-th query, and est_i is the estimated number. Note that this measure is commonly used in other experiments [10]. We also considered the average absolute error $E_{abs} = |act_i - est_i|$ and the average relative error $E_{rel} = \frac{|act_i - est_i|}{\max(1, act_i)}$ (as in [103]). They are considered for workloads derived from models M_3 and M_4, because all queries offer the same selectivity.

Method	Time ms	std.
MinSkew	34089	6781
MinSkewProg	24648	3504
rKHist	23950	2408
R-V	27434	685
FST	41321	56370

Table 6.4: Build time d=2 data sets for 1′000 buckets

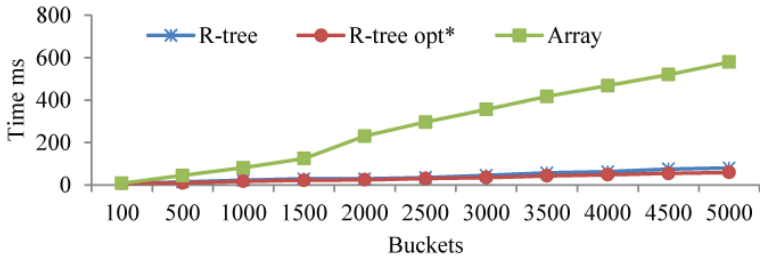


Figure 6.3: Total estimation time as a function of histogram size for the California data and M_4 query set

Build and Estimation Time

All methods except FST were able to build a 1′000 bucket histogram for 1′000′000 data objects for all data sets in less than 1 minute. Average build time in milliseconds is given in table 6.4 for d=2 data sets for 1′000 bucket histograms. The cardinality of the data sets was limited to 1′000′000 objects. The column std. shows the standard deviation. In general, the rKHist and R-V method are less sensitive to a data distribution compared to MinSkew and MinSkewProg counterparts. Recall that we construct FST histogram using random sampling of 10%. The FST method is very sensitive to data distribution, especially for non-uniform data sets. The build time for a rKhist and R-V method was dominated by external sort. The MinSkew and MinSkewProg were CPU dominated.

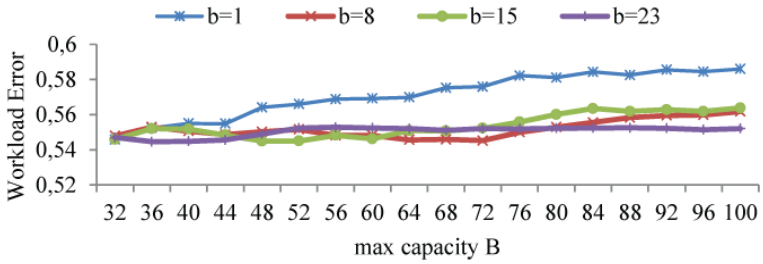


Figure 6.4: Function of E_w and capacity parameter $b = \{1, 8, 15, 23\}$ and B for the **Histogram-Generation Step** (California data, M_4)

The estimation time may become an issue if the number of histogram buckets is too high. The resulting histogram in a simple variant is represented as an array of buckets. To decrease the estimation time, histograms can be represented as main memory R-trees. Figure 6.3 depicts the function of bucket size and total workload time for different representation of a histogram. The first two are R-trees. The third one is an array of buckets. For R-tree, we used main memory setting and set the fan-out to 12 entries per node (again it was the best setting in our experiments). Additionally, we build an R-tree using histogram buckets with a *opt** partitioning method and C_V as a cost function. We constructed histograms on California data set and measured the overall estimation time of $10'000$ queries from M_4 query set with selectivity 100. We observed that if the bucket number exceeds 100, the R-tree organization displays better results.

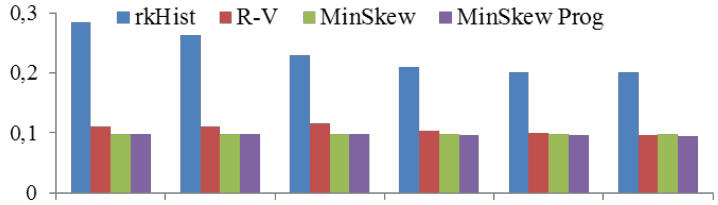
Impact of Bucket Capacity Parameter

The bucket capacity parameters $b = \max(\lfloor N_1/2m \rfloor, 1)$, $B = \lceil N_1/m \rceil + b$ in Histogram-Generation step are set depending on desired bucket number m. We examine parameter sensitivity to show that this setting displays a good accuracy. For each data set we run *opt** with different bucket capacities. Figure 6.4 shows four *opt** configurations applied on leaf nodes of the California data set generated after the Micro-Clustering. There are $N_1 = 30'398$ leaf nodes generated from $1'888'012$ rectangles. We then generate histograms for $m = 1'000$. On average, histogram buckets have capacities about $\lfloor N_1/m \rfloor = 30$ leafs. For each fixed b, we computed a E_w under query model M_4 with selectivity 100 as a function of parameter B. The values $b = 15, B = 46$ exhibit a good performance. We observed that increasing the parameter B does not lead to better histograms especially for a non-uniform data sets. In general, high $B - b$ values do not significantly improve histogram quality and even increase time complexity. In contrast, small $B - b$ values exhibit poor results for uniform and near-uniform data distributions.

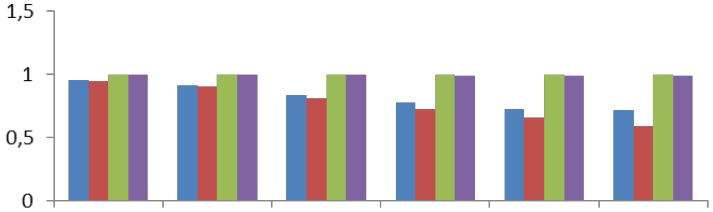
6.5.3 Experimental Results

In this section we present a detailed discussion about accuracies of different histograms. First we describe general trends observed in our experiments. Further, we discuss results obtained for small sized queries (d=2,3). We focus on M_4 workload. Subsequently, we report results for large sized queries. For the sake of brevity, we only present results of rKHist, R-V, MinSkew and MinSkewProg. Other method accuracies are presented if necessary.

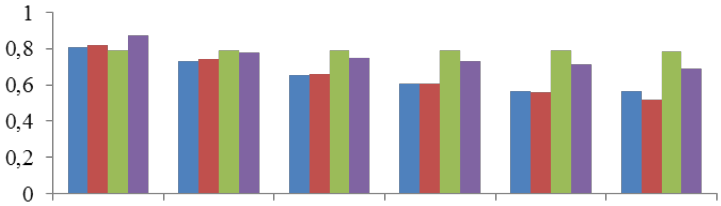
We observed several trends: first, although R-tree methods are build based on a M_1



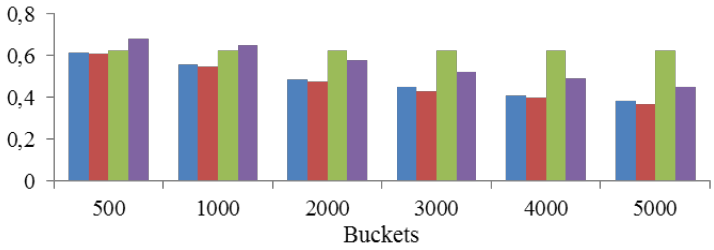
(a) abs



(b) dia



(c) par



(d) rea

Figure 6.5: E_w for rectangular data and query set M_4

model, they exhibit also good estimation results for other query workloads.

Workload M_4

Second, R-tree based methods yield better accuracies for non-uniform data distributions than MinSkew and MinSkewProg for all data and query workloads. Their selectivity

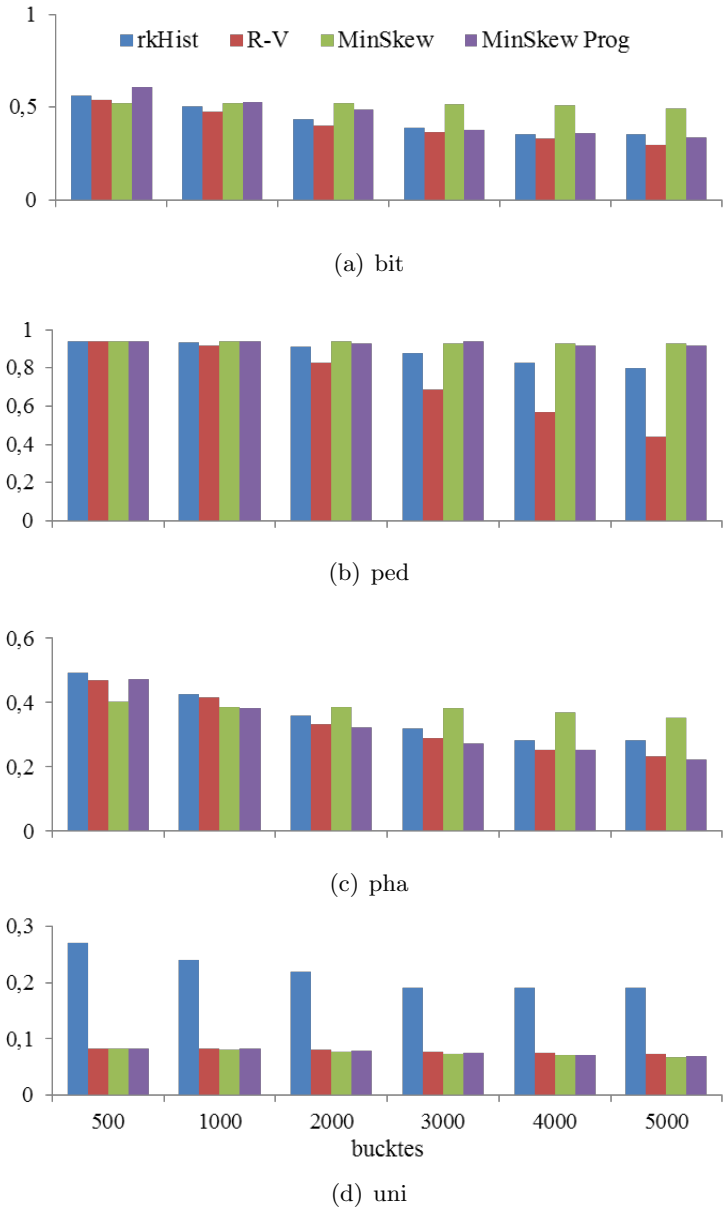


Figure 6.6: E_w for point data and query set M_4

accuracies increase more significantly with an increasing number of buckets than for MinSkew and MinsSkewProg. We also observed that with increased number of buckets, the quality of MinSkew improves marginally (as reported in [10]). In contrast, the quality of MinSkewProg increases more significantly. Using several different grid resolutions prevents MinSkewProg from allocating many buckets in a single highly skewed cluster,

since the number of buckets produced per grid is equally balanced [10]. For a large number of buckets, MinSkewProg is the better choice than MinSkew.

Third, the general deficiency of R-tree methods for uniform and near-uniform data distribution is corrected using our proposed partitioning methods. This can be explained by the fact that the produced MBRs display almost no overlap, thus, this partitioning minimizes the estimation error.

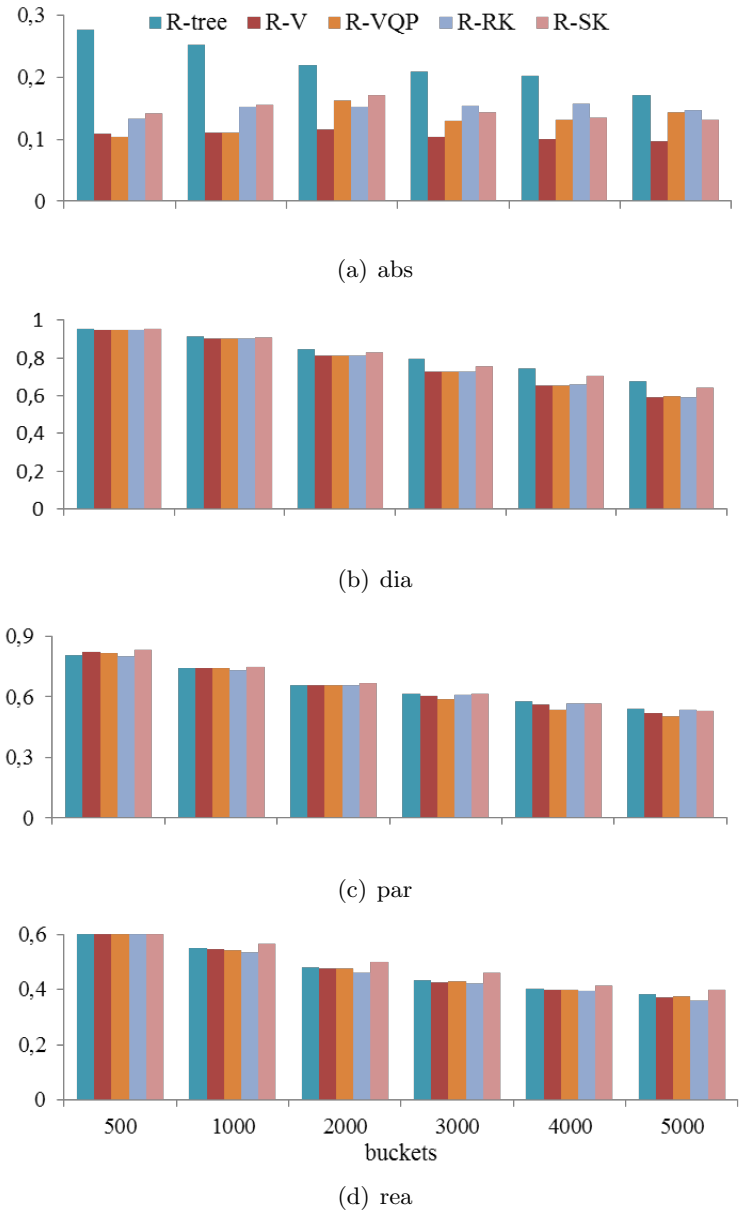
In this section, we present result for query model M_4 (query follow data distribution and query size is expressed by the number of results). Since the workload M_4 is more realistic and more difficult to handle, we report results for this model. Results for other models are discussed if necessary.

Figure 6.5 and 6.6 show results of rKhist, R-V, MinSkew and MinSkewProg for a d=2 data sets and query workload M_4 with selectivity 100. We bundle results for rectangular data sets in Figure 6.5 and for point data in Figure 6.6. Best results are achieved on *ped* data set. This data set consists of thin shaped clusters of points. Minimizing the MBR volume using dynamic programming scheme leads also to a thin shapes of MBRs, thus minimizing the estimation error. The rKHist method as well as the simple R-tree method (fixed size partitioning) have problems with uniform and near-uniform data sets. The rKHist greedy split strategy does not lead to a partitioning with small overlap introducing high estimation error. In contrast, R-V method yields better partitioning and its accuracy is comparable with MinSkew and MinSkewProg accuracies. R-V and rKHist perform better for non-uniform data sets *bit, dia, par, ped* and *rea* than MinSkew and MinSkewProg with increasing number of buckets. For *par* data set, we observed almost no difference between rKHist and R-V method. This data set has a high variance in shapes and sizes of rectangles and is difficult to handle either by R-tree histogram and index.

Figures 6.8 and 6.7 depicts results of R-tree methods compared with a fixed size partitioning strategy (R-tree) for d=2 point and rectangular data. In general, we observed that all methods using our optimized sort-partition framework display better accuracy than R-tree. Estimation accuracies of R-V, R-VQP, R-RK and R-SK do not differ significantly for non-uniform data distributions. However, C_V function exhibit better results for uniform data sets than other cost functions.

For d=3 we obtain similar results as for d=2 for all data and query sets. In general, estimation quality are slightly better for non-uniform data sets than for d=2. Figure 6.9 reports results for d=3 *rea* data set.

In Figure 6.10, we report the E_w for FST method compared with R-V and MinSkew-Prog for *rea* data set. FST Performance was very poor for all data and query sets, as



(a) abs

(b) dia

(c) par

(d) rea

Figure 6.7: E_w for rectangular data and query set M_4

we used random sampling for input data. Although applying this method on whole data set does not display better results than rKHist, R-V and MinSkewProg methods.

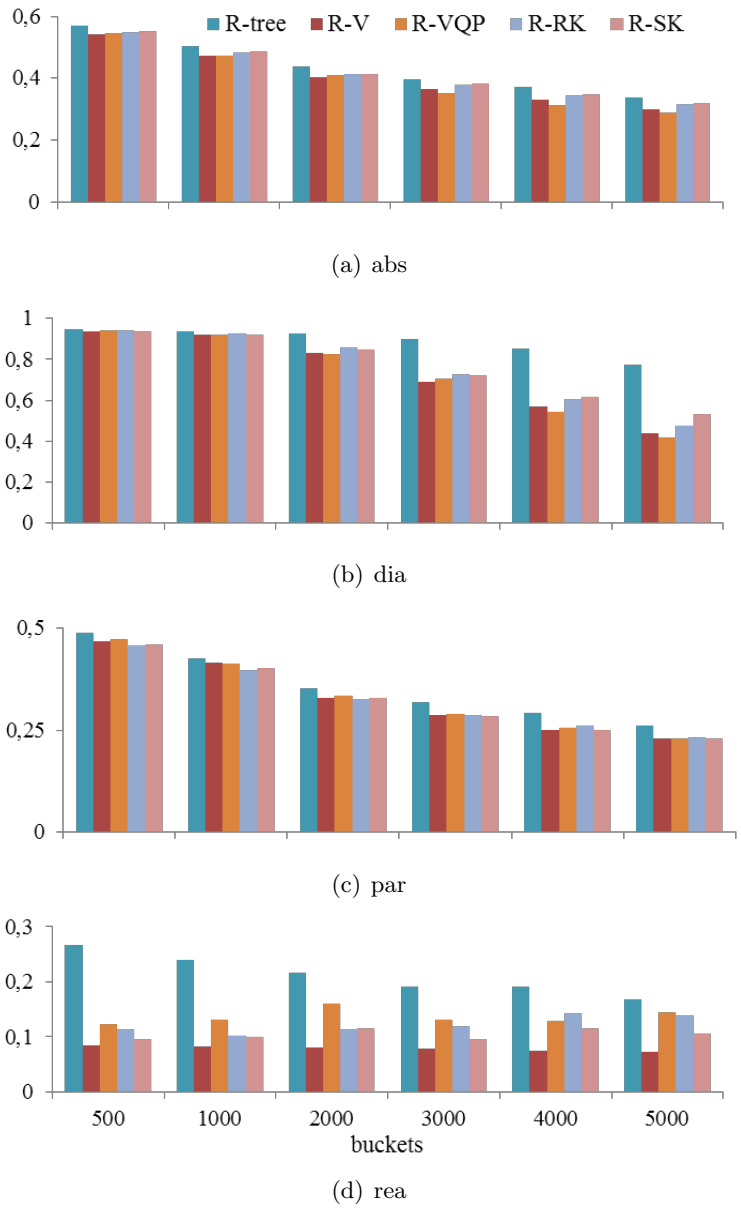


Figure 6.8: E_w for point data and query set M_4

Results for Large Queries

Figure 6.11 shows results for the California (*rea*) data set for all query workloads. For large queries R-tree based methods yield even better accuracy then MinSkew counterparts in comparison with small sized queries. Similar to small sized queries best result are achieved for non-uniform data sets. rKHist performs for two uniform *uni, abs* sets

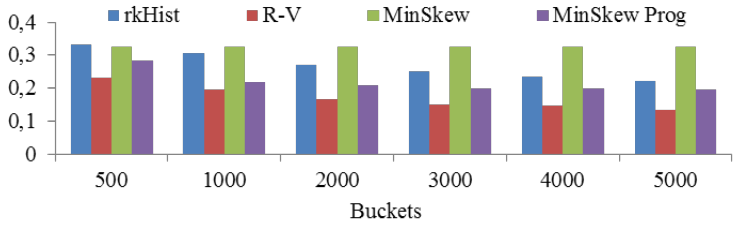


Figure 6.9: E_w for d=3 data set *rea* and query set M_4

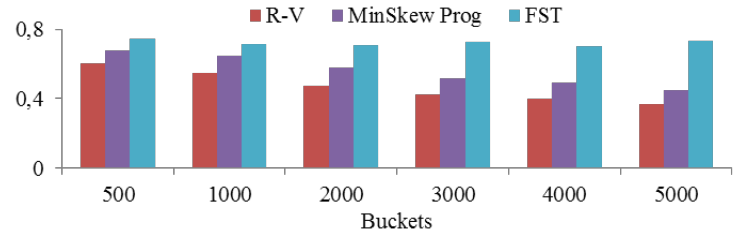
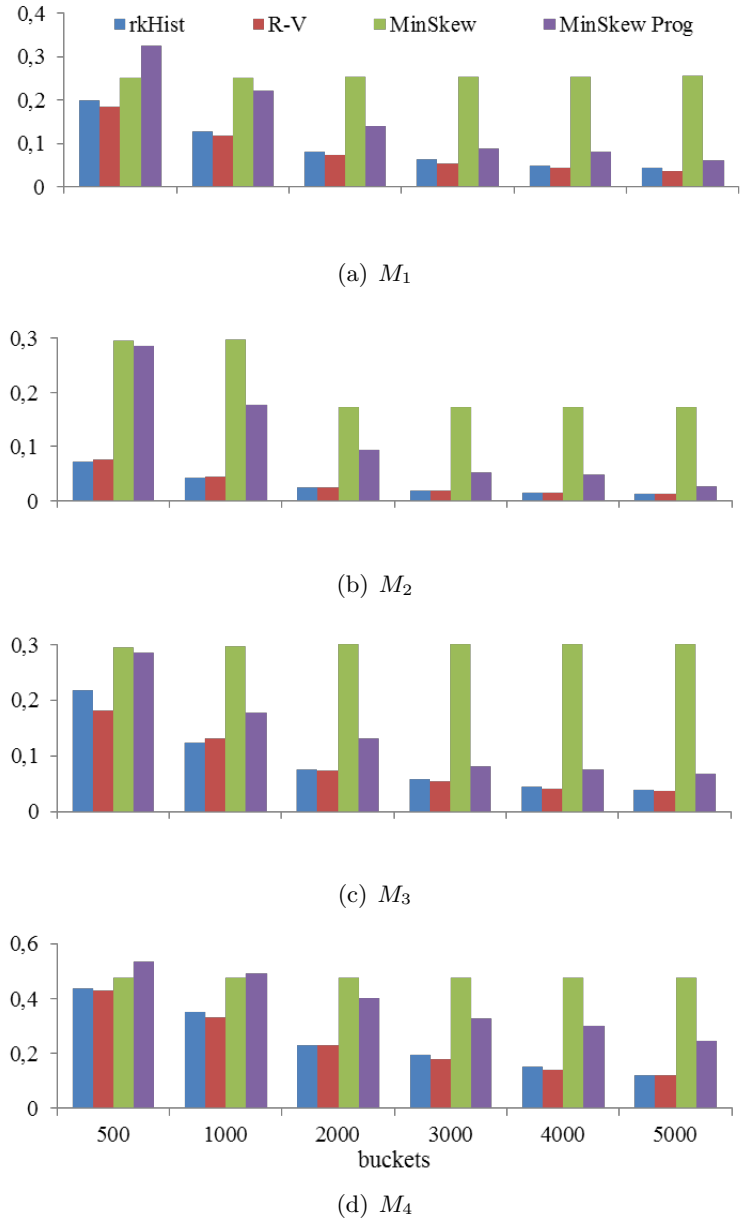


Figure 6.10: E_w for d=2 data set *rea* and query model M_4

very poor in comparison with R-V, MinSkew and MinSkewProg. Although for large queries on the California data set accuracy difference between rKHist and R-V was not that significant, with a high number of buckets R-V method was superior to other methods. Best results for R-V we achieve for synthetic data sets *abs, bit, dia, ped, pha.* Again results for *par* data set are comparable with a small sized query results. One possible solution is to partition such data distribution according to the object size and shape and construct histograms or index for each partition independently.

6.6 Conclusions

Spatial histograms are becoming increasingly important for modern GIS applications. They provide a first inexpensive view on large spatial data sets; and therefore are ideally suited for visualization and approximate query processing. In this chapter, we introduce a novel histogram method derived from a bulk-loading algorithm of R-trees. It largely eliminates the cumbersome need for setting parameters; the only ones (page capacity B and minimum occupation b) are set in the same manner as it is known for R-trees. In general, our histogram method is fairly easy to implement because it combines elementary building blocks like sorting and dynamic programming. Our method also overcomes the weak performance of R-tree histograms in the case of uniformly distributed records. Until now, it has been considered to be an open problem whether accurate R-tree histograms



(a) M_1

(b) M_2

(c) M_3

(d) M_4

Figure 6.11: E_w for the California data set; M_1, M_2 with volume 0.1 and M_3, M_4 with selectivity 1000

can be developed for uniformly distributed data. For real data sets that are known to be highly non-uniform our method generates histograms of high quality, generally much better than the ones generated by other methods.

This work also introduces a new kind of experimental setup for spatial histograms. In-

spired by cost models for spatial indexes, we consider different kind of workload scenarios rather than putting the focus only on uniformly distributed queries. This gives a more meaningful interpretation of the advantages and disadvantages of spatial histograms. In addition, we also examine the performance of histograms with a rather large number of buckets. Despite the fact of the availability of large main memories, there have been only a very few results available for histograms with more than 1000 buckets. In fact, our experiments reveal that not all of the state-of-the-art histograms can improve quality with an increasing number of buckets.

7 Conclusions and Future Work

In this work, we achieved three main results. Firstly, we showed for the first time that the loading a partial persistent B-tree is possible with I/O complexity of external sort. We developed a new kind of partial persistent B-tree that maintains all asymptotic performance guarantees of the MVBT and achieves the lower bound for loading. Secondly, we revised standard sort based bottom up loading algorithm for R-trees. We proposed the partitioning scheme that produces substantially better R-trees without increasing I/O complexity of standard sort based loading algorithms. Our algorithm improves R-trees according to the widely used cost model [73, 115, 95]. The novelty of our work is that our proposed dynamic programming algorithm considers statistical information about the query profile. We showed that knowledge about the average query side lengths and the average aspect ratio allow us to generate better R-trees. If the knowledge of the query profile is not available, we optimize R-trees according to the sum of area derived from the MBR of nodes. In both cases, we generate better R-trees than naive sort based counterparts. We also showed that for query profiles that exhibit high aspect ratio, an asymmetric Z-Curve in a combination with our partitioning scheme provides better query performance. Moreover, our preliminary experimental results with R-trees tuned using our query adaptive framework for interval data in two-dimensional space are very promising. These results show that R-trees built using the ordering derived from the average interval length exhibit good query performance. Thirdly, we showed that the proposed dynamic programming scheme allows us the generation of spatial histograms that exhibit a high accuracy and robustness for different data and query distributions. To this end, we proposed a unified framework for the R-tree bulk loading and the histogram generation.

The results in this work pose new research questions and challenges. One of the promising applications our new loading algorithm are partial persistent structures that derived from the multiversion B-tree such as the historical R-tree proposed by Tao et al. [113] or the structure for temporal range aggregates proposed by Zhang et al. [121]. In our future work, we want to investigate the combination of the bulk update approach and TMVBT proposed by Hapsaalo et al. [66]. TMVBT allows single write and multiple

read transactions at a time. Therefore, after commit the set of new live records could be inserted in a bulk rather than tuple by tuple.

The computational model that we considered for devising our MVBT$^+$ solution assumes a single disk and a single CPU. The continuously growth of historical data requires scalable data partitioning techniques over multiple disks or machines for the efficient query processing. Therefore, the design of parallel and distributed algorithms becomes more and more important. In our future work, we are interested in the developing of parallel and distributed version of our loading technique. Recently, Le et al. [79] proposed an algorithm that partitions a multiversion file in almost equi-sized partitions in the I/O complexity of external sorting. They considered a distributed case where the set of machine manages historical data. They partition an input interval set using the time dimension. By this, we can solve an offline problem in a distributed case using a combination of techniques proposed in [79] and our bulk loading technique. We first split the input using a single machine, distribute data and execute our loading technique on each machine in parallel. The approach proposed by Goodrich et al. [61] also partitions partial persistent file, yet, according to the key dimension. Therefore, we need to investigate efficient partitioning techniques for devising a solution in the distributed scenario.

Both works [79, 61] tackle an offline problem. If an MVBT has already been distributed, we need to manage new arriving data efficiently. In order to achieve a load balance, in some cases, parts of the index should be shipped to other machines [108]. Similarly, in the case of a space overflow in a current memory hierarchy, there is a demand for efficient migration policies of old data [122, 92, 89].

In general, the design of loading approaches for parallel computational models [119, 118, 109] is of interest. E.g. the computational model of Shriver and Vitter assumes D disks and P CPUs. This is a still realistic setting in modern hardware architecture. Due to the recent breakthrough in non-volatile memory technology, we need to rethink the design of external memory index structures [15], in order to improve performance in a practical setting. Algorithms designed for the cache oblivious model [55, 40] are independent from memory hardware parameters and seem to be a general solution pattern. However, not all problems can be solved in the same asymptotic I/O bounds as for the traditional I/O model [40, 11]. Among those is the problem of partial persistence search structures. Therefore, the design of loading algorithms for alternative external memory models as well as their lower I/O bounds is important for the further investigations.

We also aim to investigate alternative techniques for managing partial persistent data. Experimental evaluation of average query performance of MVBT, R-tree as well as tech-

nique proposed in [101] is also of interest. We are also interested in further developing of R-tree frameworks based on our query adaptive technique for managing interval data. Therefore, we are also interested in computing the expected average interval length analytically, based on update model.

Our query-adaptive loading strategies for R-trees highly depend on the accuracy of cost models. It is still challenging to develop better cost models that allow reorganizing indexes according to an anticipated workload in a proactive manner. Our preliminary experimental results show that a combination of spatial density histograms with our partitioning framework achieve good results for non-uniform data distributions. In order to achieve a better partitioning of the data for a given query workload, the specific way of bit shuffling in a Z-curve can also have a substantial impact on the query performance. In our future work, we are interested in further development of adaptive Z-Curve. E.g. bit shuffling could be computed in dependency of location, such that the aspect ratio from this region influence the Z-Curve orientation and symmetry. Also a combination of techniques proposed by Markl could be interesting [87]. In our future work, we are interested in the construction of such adaptive Z-curves. Further, as our proposed technique for R-tree is a heuristic (the quality depends on sorting order), designing an approximate algorithm for cost models [115, 95] is very challenging.

Our work on spatial histograms shows that R-tree based histograms derived from our partitioning framework exhibit high accuracy for different kind of data distributions. While our focus was primarily on two- and three-dimensional data, we are currently interested in the design of histograms for high-dimensional data. Similar to our design of accurate spatial histograms,the question is whether that the design principles of high-dimensional indexing can be effectively reused for high-dimensional histograms.

Bibliography

[1] Apache hbase. http://hbase.apache.org/.

[2] Ibm: A matter of time: Temporal data management in db2 for z/os. http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/.

[3] Oracle: Total recall. http://www.oracle.com/technetwork/database/application-development/total-recall-1667156.html.

[4] D. Achakeev and B. Seeger. A class of r-tree histograms for spatial databases. Technical report, Philipps-Universität Marburg, 2012.

[5] D. Achakeev and B. Seeger. A class of r-tree histograms for spatial databases. In *SIGSPATIAL*, pages 450–453, New York, NY, USA, 2012. ACM.

[6] D. Achakeev and B. Seeger. Efficient bulk updates on multiversion b-trees. In *accepted PVLDB Vol. 6 No. 14*, 2013.

[7] D. Achakeev, B. Seeger, and P. Widmayer. Sort-based query-adaptive loading of r-trees. Technical report, Philipps-Universität Marburg, 2012.

[8] D. Achakeev, B. Seeger, and P. Widmayer. Sort-based query-adaptive loading of r-trees. In *CIKM*, pages 2080–2084, New York, NY, USA, 2012. ACM.

[9] D. Achakeev, M. Seidemann, M. Schmidt, and B. Seeger. Sort-based parallel loading of r-trees. In *BigSpatial*, BigSpatial, pages 62–70, New York, NY, USA, 2012. ACM.

[10] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD '99*, pages 13–24, New York, NY, USA, 1999. ACM.

[11] P. Afshani, C. Hamilton, and N. Zeh. Cache-oblivious range reporting with optimal queries requires superlinear space. *Discrete Comput. Geom.*, 45(4):824–850, June 2011.

[12] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and r-trees with near-optimal query time. In *Proceedings of the seventeenth annual symposium on Computational geometry*, SCG '01, pages 124–133, New York, NY, USA, 2001. ACM.

[13] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[14] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB '03*, pages 81–92. VLDB Endowment, 2003.

[15] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *PVLDB*, 2(1):361–372, 2009.

[16] A. Aji, F. Wang, and J. H. Saltz. Towards building a high performance spatial query system for large scale medical imaging data. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 309–318, New York, NY, USA, 2012. ACM.

[17] D. Ajwani and H. Meyerhenke. Chapter 5. realistic computer models. In M. Müller-Hannemann and S. Schirra, editors, *Algorithm Engineering*, volume 5971 of *Lecture Notes in Computer Science*, pages 194–236. Springer Berlin Heidelberg, 2010.

[18] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala. Temporal query processing in teradata. In *EDBT*, pages 573–578, 2013.

[19] G. Antoshenkov. Random sampling from pseudo-ranked b+ trees. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 375–382, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[20] P. M. Aoki. Generalizing "search" in generalized search trees (extended abstract). In *ICDE*, pages 380–389, 1998.

[21] P. M. Aoki. How to avoid building datablades(r) that know the value of everything and the cost of nothing. *Scientific and Statistical Database Management, International Conference on*, 0:122, 1999.

[22] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract). In *WADS*, pages 334–345, 1995.

[23] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, 1996.

[24] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[25] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4:9:1–9:30, March 2008.

[26] L. Arge, A. Danner, and S.-M. Teh. I/o-efficient point location using persistent b-trees. In *ALENEX*, pages 82–92, 2003.

[27] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002.

[28] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, pages 560–, Washington, DC, USA, 1996. IEEE Computer Society.

[29] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, June 2003.

[30] L. Arge and N. Zeh. Algorithms and theory of computation handbook. chapter External-memory algorithms and data structures, pages 10–10. Chapman & Hall/CRC, 2010.

[31] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theor. Comput. Sci.*, 181:3–15, July 1997.

[32] C. Authmann. Evaluierung sortierbasierter verfahren fuer den komplettaufbau eines index, bachelorarbeit. Technical report, Philipps-Universität Marburg, 2008.

[33] B. Bamba, S. Ravada, Y. Hu, and R. Anderson. Statistics collection in oracle spatial and graph: Fast histogram construction for complex geometry objects. *PVLDB*, 6(11), 2013.

[34] B. Becker, P. G. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer. Enclosing many boxes by an optimal pair of boxes. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, pages 475–486, London, UK, 1992. Springer-Verlag.

[35] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[36] B. Becker, H.-W. Six, and P. Widmayer. Spatial priority search: An access technique for scaleless maps. In J. Clifford and R. King, editors, *SIGMOD '91*, pages 128–137. ACM Press, 1991.

[37] L. Becker, H. Partzsch, and J. Vahrenhold. Query responsive index structures. In *GIScience '08*, pages 1–19, 2008.

[38] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90*, pages 322–331, New York, NY, USA, 1990. ACM.

[39] N. Beckmann and B. Seeger. A revised r*-tree in comparison with related index structures. In *SIGMOD '09*, pages 799–812. ACM, 2009.

[40] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *FOCS*, pages 399–409, 2000.

[41] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *SPAA*, pages 81–92, 2007.

[42] J. V. d. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. Xxl - a library approach to supporting efficient implementations of advanced database queries. In *VLDB '01*, pages 39–48, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[43] J. V. d. Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *VLDB '01*, pages 461–470, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[44] J. V. d. Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB '97*, pages 406–415, 1997.

[45] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[46] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303 – 320, 1980.

[47] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: a multidimensional workload-aware histogram. *SIGMOD Rec.*, 30:211–222, May 2001.

[48] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM 2009*, pages 302–319, Berlin, Heidelberg, 2009. Springer-Verlag.

[49] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[50] J. V. den Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *VLDB*, pages 168–179, 1996.

[51] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *VLDB '94*, pages 558–569, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[52] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[53] T. Eavis and A. Lopez. Rk-hist: an r-tree based histogram for multi-dimensional selectivity estimation. In *CIKM '07*, pages 475–484, New York, NY, USA, 2007. ACM.

[54] H. Edelsbrunner. A new approach to rectangle intersections part i. *International Journal of Computer Mathematics*, 13(3-4):209–219, 1983.

[55] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[56] F. Furfaro, G. M. Mazzeo, D. Saccà, and C. Sirangelo. Hierarchical binary histograms for summarizing multi-dimensional data. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 598–603, New York, NY, USA, 2005. ACM.

[57] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.

[58] Y. J. García R, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *GIS '98*, pages 163–164, New York, NY, USA, 1998. ACM.

[59] Y. Giora and H. Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5:28:1–28:51, July 2009.

[60] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD*, pages 847–854, 2009.

[61] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry (preliminary version). In *FOCS*, pages 714–723, 1993.

[62] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, Mar. 2006.

[63] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[64] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. *SIGMOD Rec.*, 29:463–474, May 2000.

[65] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM.

[66] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen. Transactions on the multiversion b+-tree. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1064–1075, New York, NY, USA, 2009. ACM.

[67] T. Haapasalo, I. Jaluta, S. Sippu, and E. Soisalon-Soininen. On the recovery of r-trees. *IEEE Trans. Knowl. Data Eng.*, 25(1):145–157, 2013.

[68] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *J. ACM*, 49(1):35–55, Jan. 2002.

[69] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

[70] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[71] Y. Ioannidis. The history of histograms (abridged). In *VLDB '2003*, pages 19–30. VLDB Endowment, 2003.

[72] H. V. Jagadish, V. Poosala, N. Koudas, K. Sevcik, S. Muthukrishnan, and T. Suel. Optimal histograms with quality guarantees. In *In VLDB*, pages 275–286, 1998.

[73] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM '93*, pages 490–499, New York, NY, USA, 1993. ACM.

[74] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 257–276, London, UK, UK, 1999. Springer-Verlag.

[75] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: a unified data structure for processing queries on temporal data in sap hana. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1173–1184, New York, NY, USA, 2013. ACM.

[76] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 134–145, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[77] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 62–72, New York, NY, USA, 1997. ACM.

[78] K. Kulkarni and J.-E. Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, Oct. 2012.

[79] W. Le, F. Li, Y. Tao, and R. Christensen. Optimal splitters for temporal and multi-version databases. In *SIGMOD*, 2013.

[80] S. Leutenegger, M. A. Lopez, and J. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.

[81] D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[82] D. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, pages 315–324, 1989.

[83] D. B. Lomet. Grow and post index trees: Roles, techniques and future potential. In O. Günther and H.-J. Schek, editors, *SSD*, volume 525 of *Lecture Notes in Computer Science*, pages 183–206. Springer, 1991.

[84] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal db: transaction time support for sql server. In *SIGMOD Conference*, pages 939–941, 2005.

[85] D. B. Lomet, M. Hong, R. V. Nehme, and R. Zhang. Transaction time indexing with version compression. *PVLDB*, 1(1):870–881, 2008.

[86] D. B. Lomet and F. Li. Improving transaction-time dbms performance and functionality. In *ICDE*, pages 581–591, 2009.

[87] V. Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*, volume 59 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1999.

[88] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. *SIGMOD Rec.*, 17:28–36, June 1988.

[89] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the lham log-structured history data access method. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 452–463, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[90] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *ICDT '99*, pages 236–256, London, UK, 1999. Springer-Verlag.

[91] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM.

[92] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[93] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84*, pages 181–190, New York, NY, USA, 1984. ACM.

[94] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.

[95] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *PODS '93*, pages 214–221, New York, NY, USA, 1993. ACM.

[96] B.-U. Pagel, H.-W. Six, and M. Winter. Window query-optimal clustering of spatial objects. In *PODS '95*, pages 86–94, New York, NY, USA, 1995. ACM.

[97] A. Papadopoulos and Y. Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Comput.*, 29(10):1419–1444, 2003.

[98] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25:294–305, June 1996.

[99] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.

[100] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB '97*, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[101] S. Ramaswamy. Efficient indexing for constraint and temporal databases. In *Proceedings of the 6th International Conference on Database Theory*, ICDT '97, pages 419–431, London, UK, UK, 1997. Springer-Verlag.

[102] O. Rodeh. B-trees, shadowing, and clones. *TOS*, 3(4), 2008.

[103] Y. J. Roh, J. H. Kim, Y. D. Chung, J. H. Son, and M. H. Kim. Hierarchically organized skew-tolerant histograms for geographic data objects. In *SIGMOD '10*, pages 627–638, New York, NY, USA, 2010. ACM.

[104] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD Conference*, pages 17–31, 1985.

[105] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999.

[106] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[107] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[108] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 765–778, New York, NY, USA, 2008. ACM.

[109] N. Sitchinava and N. Zeh. A parallel buffer tree. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 214–223, New York, NY, USA, 2012. ACM.

[110] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[111] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *ICDE '06*, pages 39–, Washington, DC, USA, 2006. IEEE Computer Society.

[112] A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[113] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.

[114] Y. Tao and D. Papadias. Adaptive index structures. In *VLDB '02*, pages 418–429, 2002.

[115] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *PODS '96*, pages 161–171, New York, NY, USA, 1996. ACM.

[116] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Trans. Knowl. Data Eng.*, 9(3):391–409, 1997.

[117] R. R. Vatsavai, A. Ganguly, V. Chandola, A. Stefanidis, S. Klasky, and S. Shekhar. Spatiotemporal data mining in the era of big spatial data: algorithms and applications. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '12, pages 1–10, New York, NY, USA, 2012. ACM.

[118] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[119] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[120] K. Yi, X. Lian, F. Li, and L. Chen. The world in a nutshell: Concise range queries. *IEEE Trans. Knowl. Data Eng.*, 23(1):139–154, 2011.

[121] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM Trans. Database Syst.*, 33(2):12:1–12:39, June 2008.

[122] R. Zhang and M. Stradling. The hv-tree: a memory hierarchy aware version index. *PVLDB*, 3(1):397–408, 2010.